MASTER EN SCIENCES INFORMATIQUES

Débogage de modèles B événementiels en utilisant le plugin ProB disprover

Ligot, Olivier

Award date:
2007

Awarding institution:
Universite de Namur

FUNDP - Namur
Institut d'Informatique

Année académique 2006-2007

Débogage de modèles B événementiels en utilisant le plugin ProB *disprover*

Olivier Ligot

Mémoire présenté en vue de l'obtention du grade de maître en informatique

Résumé

Le B classique, comme son sucesseur le B événementiel, sont toutes les deux des méthodes formelles utilisées pour le développement de systèmes informatiques dans lesquels l'exactitude est prouvée formellement. Cependant, plus la spécification devient complexe, plus des obligations de preuve doivent être *déchargées*. Bien que beaucoup de ces obligations de preuve peuvent être *déchargées* automatiquement par des outils tels que la plateforme RODIN, il en reste tout de même un nombre considérable à prouver interactivement. Cela peut être du à une preuve requise trop compliquée, ou à un modèle B erroné. Dans ce mémoire, nous décrivons un *plugin disprover* pour RODIN qui utilise l'animateur et *model checker* ProB pour trouver automatiquement des contre-exemples à une obligation de preuve problématique. Dans le cas où le *disprover* trouve un contre-exemple, l'utilisateur peut directement examiner la source du problème (comme mentionnée par le contre-exemple) et ne doit pas essayer de prouver l'obligation de preuve. Nous discutons également sous quelles conditions notre *plugin* peut être utilisé comme *prover*, c'est-à-dire quand l'absence de contre-exemples est en fait une preuve de l'obligation de preuve.

Mots-clés : RODIN, ProB, B événementiel, B classique, *prover* automatique.

Abstract

The B-method, as well as its offspring Event-B, are both tool-supported formal methods used for the development of computer systems whose correctness is formally proven. However, the more complex the specification becomes, the more proof obligations need to be discharged. While many proof obligations can be discharged automatically by recent tools such as the RODIN platform, a considerable number still have to be proven interactively. This can be either because the required proof is too complicated or because the B model is erroneous. In this thesis we describe a disprover plugin for RODIN that utilizes the ProB animator and model checker to automatically find counterexamples for a given problematic proof obligation. In case the disprover finds a counterexample, the user can directly investigate the source of the problem (as pinpointed by the counterexample) and should not attempt to prove the proof obligation. We also discuss under which circumstances our plug-in can be used as a prover, i.e., when the absence of a counterexample actually is a proof of the proof obligation.

Keywords: RODIN, ProB, Event-B, B-Method, Autoprover.

Je tiens à remercier toutes les personnes qui m'ont aidé dans la rédaction de ce mémoire et lors du stage. Je remercie Monsieur Michael Leuschel et Jens Bendisposto pour leur accueil à Düsseldorf, leurs conseils dans la rédaction de l'article et leurs conseils précieux lors du développement du *plugin*. Je remercie particulièrement mon promoteur, Monsieur Wim Vanhoof, pour ses conseils avisés et son soutien permanent sans lesquels ce travail ne serait pas ce qu'il est. En outre, je remercie mes parents et ma copine Céline pour leurs encouragements constants.

Table des matières

Liste des tableaux

Liste des définitions

Chapitre 1

Introduction

1.1 Contexte du travail

De nos jours, la construction et la validation de spécifications sont de mise. Diverses méthodes actuelles permettent de représenter un problème avec un niveau d'abstraction élevé. Une fois le problème modélisé, il devrait en principe être validé tant d'un point de vue organisationnel (par les développeurs, voire les clients) que logique (exactitude, justesse).

Dans ce mémoire, nous nous intéressons plus particulièrement à deux de ces méthodes formelles, à savoir le langage B classique et son successeur le B événementiel. Ces deux langages s'accompagnent, en plus d'une syntaxe et d'une sémantique formelle, d'outils de modélisation et de validation de preuves. Grâce à ces techniques de validation, il est possible d'affirmer l'exactitude d'une ou de plusieurs spécifications formelles.

Le processus de modélisation se déroule en plusieurs étapes. Dans un premier temps, le problème est spécifié à l'aide d'un formalisme de niveau d'abstraction très élevé. Dans un second temps, la spécification est transformée en une suite de raffinements, chaque raffinement apportant une touche de plus bas niveau à la spécification. Pour la spécification, et pour chacun des raffinements, il est possible de prouver la justesse de la description. Plus précisément, la preuve consiste à vérifier certaines caractéristiques qui sont nécessaires pour que la spécification soit valide. La validation d'une spécification permet dès lors d'acquérir un niveau de confiance dans le développement du système informatique. La dernière de ces transitions est l'implémentation à proprement parler qui peut être directement exécutée sur une machine. Si la spécification s'est avérée prouvée à chacune des différentes étapes de construction, alors l'implémentation reflète d'une part la première description de haut niveau défini dans la première phase et d'autre part l'exactitude d'un point de vue formel.

1.2 Problématique

Pour effectivement prouver la spécification que nous construisons, nous avons besoin d'un mécanisme, appelé *prover*, qui nous guide dans la preuve. De manière informelle, ce *prover* prend comme entrée une partie de la spécification, traite cette entrée et fournit en sortie le résultat de la preuve, généralement sous forme affirmative ou négative. Cependant, il existe certains cas dans lequels la preuve d'une spécification n'est pas directement réalisable. Cela peut par exemple être le cas d'une spécification erronée mais dont l'erreur est difficilement détectable par l'utilisateur.

C'est dans cette optique que nous introduisons dans ce mémoire un *disprover* qui, à l'inverse du *prover*, tente non pas de prouver une spécification mais plutôt de trouver un contre-exemple qui prouve que la spécification est erronée. Ce *disprover* est construit pour le langage B événementiel, et fournit un *plugin* pour l'outil principal de modélisation du langage appelé RODIN. Le développement d'un tel plugin soulève plusieurs questions : quel est l'apport d'un *disprover* par rapport à un *prover* classique ? Si le *disprover* rend compte de cas particuliers non détectables par le *prover*, quelles sont les limites du *disprover* ?

L'une des premières phases du *disprover plugin* est de transformer la spécification écrite en B événementiel en B classique, ce dernier étant effectivement utilisé pour la recherche du contre-exemple. Dans notre approche, la transformation est relativement bien délimitée mais on pourrait se demander quelles sont les limites de telles transformations du langage B événementiel en B classique. Le B événementiel étant le successeur du B classique, nous avons envisagé la transformation dans un sens, mais qu'en est-il de la transformation dans l'autre sens ? Est-elle pertinente ?

1.3 Contenu du mémoire

Le mémoire est structuré de la manière suivante :

- La **partie I** est exclusivement réservée au *B classique* [Abr96]. Le chapitre 2 décrit d'un point de vue théorique un sous-ensemble du langage. Des exemples sont fournis afin d'illustrer les concepts fondamentaux. L'annexe A rappelle brièvement quelques définitions mathématiques. Le chapitre 3 est divisé en trois parties : dans un premier temps, le mécanisme de preuve du B classique est introduit. Ce mécanisme est ensuite appliqué aux exemples vus dans le chapitre 2. Finalement, la dernière partie est consacrée à la présentation de l'animateur et model checker ProB [LB03].
- La **partie II** de ce mémoire se réfère au *B événementiel* [ROD05b]. Le chapitre 4 développe les concepts théoriques du langage. Des exemples sont apportés afin de mettre en lumière les notions fondamentales. Le

chapitre 5 souligne et traite dans un premier temps de l'importance de la notion de preuve. Dans un deuxième temps, les exemples du chapitre 4 sont prouvés. Finalement, la plateforme de développement RODIN [Rom06] est exposée.

– La **partie III** se compose du chapitre 6. Celui-ci détaille l'architecture globale ainsi que les choix d'implémentation du *disprover plugin*.

Origine des chapitres

Le chapitre 6 est basé sur un article publié lors de la conférence [LBL07].

Première partie

Le B classique

Chapitre 2

Introduction au B classique

La méthode B est une approche formelle à la spécification et au développement de systèmes informatiques. C'est un langage qui est basé sur la notion de machine abstraite – **A**bstract **M**achine **N**otation ou AMN – qui fournit un framework commun pour la construction de spécifications, raffinements et implémentations [Sch01]. La spécification est la première phase de modélisation d'un système. Une fois celle-ci finie, on peut raffiner par une série d'étapes, le modèle dans le but d'aboutir finalement à une implémentation. La méthode B est surtout utilisée dans des situations critiques où l'accent est mis sur la preuve et la validité. Il est en effet possible, et ce à chaque étape du développement, de prouver de manière formelle la validité du système. Elle a notamment été appliquée avec succès lors de la construction de la ligne de métro 14 à Paris [BDMB98].

Ce chapitre s'inspire en partie de [Sch01]. Le langage étant assez vaste, nous ne nous étalerons pas sur celui-ci mais présenterons plutôt ses concepts importants nous aidant dans la réalisation du mémoire. Le lecteur désireux d'approfondir ses connaissances dans le domaine peut se reporter à [Sch01].

La structure du chapitre est la suivante : dans un premier temps, la notion de machine abstraite est explicitée ; des propriétés mathématiques du langage sont ensuite décrites ; un exemple est ensuite apporté pour illustrer les concepts ; finalement, le non déterminisme est abordé.

Remarques préliminaires

Il existe 2 notations différentes – mathématique et `ASCII` – pour la syntaxe du B classique. Nous utiliserons dans ce chapitre la représentation sous forme mathématique. La représentation sous forme `ASCII` sera quand à elle utilisée dans le chapitre 3 concernant ProB.

2.1 Machine Abstraite

Le but général de la méthode B est de modéliser, dans un formalisme donné, un (sous) problème dans lequel la validité de la solution est primordiale. Pour ce faire, il est d'abord indispensable de passer par une première phase de spécification, où l'on décrit les fonctionnalités souhaitées du système (le *quoi*) sans forcément donner la façon de procéder (le *comment*). La notation utilisée pour représenter le problème est la *machine abstraite*, aussi appelée **A**bstract **M**achine **N**otation, ou AMN. Donnons dans un premier temps une définition de celle-ci.

Définition 2.1 Machine abstraite

Une *machine abstraite* est un pseudo-langage de programmation composé :

1. d'un nom ;
2. d'un *état* local ;
3. d'un ensemble d'*opérations* qui peuvent accéder et mettre à jour l'état.

Dans le contexte du B classique, une machine abstraite est composée d'un ensemble de clauses dont la structure générale est donnée au listing 2.1. Certaines clauses sont obligatoires, d'autres facultatives. Comme tout autre langage, le B a une syntaxe et une sémantique. Plutôt que de décrire celles-ci en détail, nous nous attarderons dans les sous-sections qui suivent à détailler la structure de chaque clause, en soulignant les propriétés remarquables de chacunes.

Listing 2.1 – Structure générale d'une machine abstraite

```
1  MACHINE  N(p)
2  CONSTRAINTS  C
3  SETS  St
4  CONSTANTS  k
5  PROPERTIES  B
6  VARIABLES  v
7  INVARIANT  I
8  INITIALISATION  T
9  OPERATIONS
10    y ← op(x) =
11      PRE  P
12      THEN  S
13      END;
14    ...
15 END
```

2.1.1 Paramètres et contraintes

Une machine porte toujours un nom, qui est unique[1]. Elle peut éventuellement être accompagnée de paramètres. Ceux-ci sont alors directement spécifiés derrière le nom. Les paramètres ajoutent une dimension dynamique supplémentaire à la machine, dans le sens où une autre machine peut utiliser la première en fixant les paramètres.

Les paramètres, comme d'autres constructions en B, doivent toujours être accompagnés d'un type. Il existe un ensemble de types prédéfinis, mais rien n'empêche de définir ses propres types. Remarquons que contrairement à ce que nous pourrions croire, $\mathbb{N}$ n'est pas un type[2] : une variable appartenant à $\mathbb{N}$ aura dès lors le type $\mathbb{Z}$. Le type des paramètres, et éventuellement des contraintes additionnelles sur ces paramètres, sont spécifiés dans la clause `CONSTRAINTS` de la machine. Donnons un exemple pour illustrer ces concepts.

Exemple 2.1. La machine suivante porte le nom *Parking* et a un paramètre, *voiture* qui est un naturel positif supérieur ou égal à 50. Bien que *voiture* soit un naturel, son type est $\mathbb{Z}$.

```
1  MACHINE Parking(voiture)
2  CONSTRAINTS
3     voiture∈ℕ1 ∧ voiture≥50
```

2.1.2 Ensembles

Le B usant du langage mathématique en abondance, il n'est pas étonnant de pouvoir définir des ensembles. La définion de ceux-ci en B classique est identique à celle en mathématique : une collection d'objets bien définis tel que pour tout objet x quelconque, soit x fait partie de la collection, soit x n'en fait pas partie. Une distinction est cependant apportée quant au type d'ensemble considéré :

Définition 2.2 Ensemble énuméré/différé

Un ensemble est *énuméré* si tous ses éléments sont présents dans sa définition. Un ensemble non énuméré est dit *diferré*[3]

La définition d'un ensemble énuméré donne tous ses éléments, sans exception. Les ensembles sont définis dans la clause `SETS` de la machine.

Exemple 2.2. Une guitare est soit *classique*, soit *acoustique*, soit *electrique*.

[1] Le nom d'une machine est unique par rapport aux autres machines qui sont en relation avec elle.

[2] il en va de même pour des langages tels que C ou Java

[3] le terme anglais *'deferred'* est fort usité dans la littérature

```
1  SETS
2    GUITARES = {classique, acoustique, electrique}
```

2.1.3 Constantes et propriétés

Une constante est une variable dont la valeur ne peut changer. A l'instar des paramètres, un type doit toujours être associé à une constante. Ce type, ainsi que d'éventuelles propriétés additionnelles sur les constantes sont spécifiés dans la clause `PROPERTIES` de la machine.

Exemple 2.3. Soit un auditoire avec un nombre limité de places : 100. Une manière de modéliser cette limite est de définir une constante *places* telle que *places* est un nombre naturel, inférieur ou égal à 100. La remarque concernant le type des nombres naturels est toujours valable pour les constantes : le type de *places* est donc $\mathbb{Z}$.

```
1  CONSTANTS
2    places
3  PROPERTIES
4    places∈ℕ ∧ places≤100
```

2.1.4 Variables

A n'importe quel moment, une machine abstraite se trouve dans un état spécifique. Une manière de faire passer la machine d'un état dans un autre est de définir un ensemble de variables, appelées *variables d'états* auxquelles sont associées des valeurs initiales, puis de modifier les valeurs de ces variables au fur et à mesure. Chaque variable porte un type. Les variables sont définies dans la clause `VARIABLES` de la machine. Seul le nom, et non le type, apparait dans cette clause. Donnons un exemple de définition de deux variables.

Exemple 2.4. Définition de deux variables *e* et *sub*, dont le type sera donné dans la sous-section 2.1.5.

```
1  VARIABLES
2    e, sub
```

2.1.5 Invariant

A l'opposé des langages traditionnels dans lesquels l'invariant porte sur une partie du programme (traditionnellement boucle `for` ou `while`), il porte ici sur la totalité de la machine. L'invariant fixe dans un premier temps le type de **toutes** les variables présentes dans la machine. A ces types peuvent

être ajouté des conditions particulières sur les variables : borne supérieure, relation entre plusieurs variables, ...

L'invariant est défini dans la clause `INVARIANT` de la machine. Il est de temps à autre appelé la *spécification statique* de la machine. La propriété suivante doit être vérifiée pour chaque état atteint par la machine :

Propriété 2.1 Consistance/inconsistance d'une machine

Une machine est *consistance* si, à l'initialisation, et pour l'exécution de chaque opération, l'invariant tient. A l'opposé, une machine est *inconsistance* si elle viole l'invariant.

Grâce à la consistance, on garantit que l'initialisation est un état cohérent et que, si l'on se trouve dans un tel état avant l'exécution d'une opération, alors on restera dans un état cohérent après l'exécution. L'utilisation du 'model checking' permet, par un parcours de l'espace des états, une recherche des 'deadlocks' et le viol de l'invariant.

Reprenons l'exemple 2.4 et donnons l'invariant des deux variables.

Exemple 2.5. La variable e est comprise dans l'intervalle $[0 \dots 2]$ et a comme type $\mathbb{Z}$ car elle prend comme valeur des entiers. La variable *sub* est un sous-ensemble strict de l'ensemble des parties[4] des nombres naturels positifs. **TODO : donner le type de** *sub*

```
1   INVARIANT
2     e ∈ 0 ... 2
3     sub ⊂ ℙ(ℕ1)
```

2.1.6 Initialisation

A toute variable est associée une valeur initiale, que l'on spécifie dans la clause `INITIALISATION` de la machine. Cette initialisation est nécessaire pour "amorcer" la machine. Pour chaque variable, sa valeur initiale doit être en accord avec son type, sinon la machine est considérée comme incohérente.[5]

2.1.7 Opérations

Les opérations sont ce qu'on appelle la *spécification dynamique* de la machine : elles spécifient comment la machine évoluera dans le temps, par des modifications qui auront comme effet un changement d'état. Ce changement d'état s'effectue, à l'inverse de la majorité des langages, en une étape[6] (i.e. aucune séquence, ni boucle n'est permise). Dans le cas d'une modification de

[4] couramment appelé 'power set' : l'ensemble de tous les sous-ensembles

[5] Nous verrons ultérieurement comment les logiciels de développement du langage B prennent en charge les erreurs d'une machine.

[6] Cette propriété est valable pour une spécification, pas pour une implémentation.

valeurs, les variables ont des valeurs avant l'exécution d'une opération, puis reçoivent éventuellement des autres valeurs **en parallèle** après l'exécution de l'opération[7].

La définition d'une opération est :

Définition 2.3 Opération

Une *opération* est munie des informations suivantes :

1. un nom
2. un corps
3. un (des) paramètre(s) d'entrée (optionnel)
4. un (des) paramètre(s) de sortie (optionnel)
5. une précondition (optionnel)

Eclairons cette définition. Toute opération contient un nom qui est unique par rapport aux noms des autres opérations de la machine. Elle peut inclure des paramètres d'entrée et/ou de sorties. Ces paramètres influencent dès lors le comportement, plus communément appelé *corps* de l'opération.

L'exemple 2.6 traite des entrées/sorties des opérations.

Exemple 2.6. L'opération *manger* prend comme entrées les variables i_1, i_2, i_3 et produit en sortie j.

$$j \leftarrow manger(i_1, i_2, i_3)$$

L'opération *test* a une entrée *var* et aucune sortie.

$$test(var)$$

L'opération *calcul* n'a pas d'entrée et produit *reponse* et *donnees* comme sortie.

$$reponse, donnees \leftarrow calcul$$

Remarquons qu'une opération peut, syntaxiquement parlant, n'avoir aucune entrée ni aucune sortie. Son utilité est toutefois réduite.

Préconditions

La précondition est un prédicat qui définit, le cas échéant, des restrictions sur les paramètres d'entrée. Dans le cas où des paramètres sont fournis en entrée, il devient obligatoire de définir dans celle-ci le type de **tous** les paramètres d'entrée. Les paramètres de sortie n'ont pas de type au sens

[7] voir aussi l'exemple 4.1

propre, mais sont tout de même régis par des contraintes (voir le chapitre 3 pour des explications). Une opération est dite active si sa précondition est vérifiée ; dans ce cas, le corps de l'opération pourra être exécuté. Si la précondition ne peut pas être vérifiée, l'opération se trouve dans un état où elle n'est pas active. Clarifions le tout par un petit exemple.

Exemple 2.7. Reprenons l'exemple 2.6, et plus particulièrement l'opération $manger(i_1, i_2, i_3)$. Soit la précondition de cette opération

$$i_1 < 4 \wedge (i_2 - i_3) * i_1 \geq 0$$

La précondition est vérifiée si les paramètres d'entrée ont pour valeur : $i_1 = 2$, $i_2 = 5$ et $i_3 = 5$. On peut dès lors s'attaquer au corps de l'opération. Par contre, l'opération n'est pas active si, par exemple, $i_1 = 1$, $i_2 = 2$ et $i_3 = 3$.

Corps

Le corps est le coeur de l'opération : il décrit ce qui est réalisé – quelles sont les modifications apportées aux variables et/ou paramètres – une fois l'éventuelle précondition vérifiée. Le corps d'une opération peut être vide (i.e. en utilisant la notation `skip`) : même si cela peut sembler à priori insensé, son utilisation peut se révéler utile, comme nous le verrons dans le chapitre 6 traitant du 'disprover plug-in'. Généralement, le corps permettra soit de produire une valeur au(x) paramètre(s) de sortie[8], soit de modifier la valeur des variables en utilisant une assignation.

Assignation

Outre retourner une valeur, le corps d'une opération peut assigner de nouvelles valeurs aux variables de la machine. Il existe 2 types d'assignations : la simple et la multiple dont les syntaxes sont relativement :

$$x := E \tag{2.1a}$$

$$x, \ldots, y := E, \ldots, F \tag{2.1b}$$

Dans (2.1a), la valeur de l'expression E est calculée, pour ensuite écraser l'ancienne valeur de la variable x. Dans (2.1b), toutes les expressions $E, \ldots, F$ sont évaluées, ensuite le résultat de chacune est assigné **simultanément** aux variables correspondantes. Les variables qui sont mises à jour doivent être distinctes[9].

Il existe une deuxième notation sémantiquement identique à la première pour l'assignement multiple de variables : $\|$. Terminons cette partie consacrée aux opérations par un dernier exemple.

[8] Notons que ce point est obligatoire en présence de paramètre(s) de sortie

[9] Cela peut être problématique pour les tableaux, mais c'est un cas que nous ne traitons pas ici.

Exemple 2.8. Echange – swap – des variables x et y. Les équations (4.3j) et (2.2b) sont sémantiquement équivalentes : il est donc possible de remplacer l'une par l'autre dans une machine et vis-versa. Rappelons encore une fois que cet échange s'opère instantanément.

$$x, y := y, x \tag{2.2a}$$

$$x := y \parallel y := x \tag{2.2b}$$

2.2 Propriétés mathématiques

Comme brièvement présenté dans la section précédente, le langage B manipule des éléments mathématiques, tels que des nombres naturels, des ensembles, des 'powersets', ... Il est néanmoins envisageable de profiter d'un nombre conséquent de propriétés mathématiques : relations, fonctions, inverse, maximum/minimum, modulo, ... Une bonne partie de ces propriétés sont exposées dans cette section. Un rappel des définitions mathématiques élémentaires se trouve en annexe A. Les définitions qui suivent sont directement tirées de [Cle02].

2.2.1 Relation

La relation est une notion essentielle dans le cadre de la méthode B. Sa définition et quelques propriétés sont rappelées dans l'annexe A.1.1. Le B se restreint aux relations d'un ensemble dans un autre. Si X et Y sont deux ensembles, on note $X \leftrightarrow Y$ leur relation. Il est également possible de décrire une relation en fonction de couples d'éléments ; deux notations équivalentes[10] sont à notre disposition : (s, t) et $s \mapsto t$, où s appartient au premier ensemble et t au second. L'emploi de restrictions définies ci-dessous, et le fait qu'une fonction est un cas particulier d'une relation[11] impliquent que le concept de relation est fort usité dans B.

Restrictions

Les restrictions suivantes peuvent être amenées à la fois sur le domaine et sur le rang d'une relation. Rappelons que si E est un ensemble, alors la notation $\mathbb{P}(E)$, ou 'powerset' de E, désigne l'ensemble de tous les sous-ensembles de E.

Dans la définition précédente concernant la surcharge par la droite, les éléments de r_2 notés $(x \mapsto z)$ surchargent les éventuels éléments $(x \mapsto y)$ de r_1. La surcharge peut de façon équivalente être écrite sous la forme :

$$r_1 \lhd\!\!\!+ r_2 = r_2 \cup (\mathrm{dom}(r_2) \lhd r_1)$$

[10] Bien qu'équivalentes, certains logiciels n'acceptent qu'une des deux notations.

[11] Ce n'est pas vrai pour tous les cas : voir l'annexe A.1.3 pour de plus amples informations.

Définition 2.4 Restriction/soustraction sur le domaine

Soit trois ensembles X, Y et E, de types respectifs $\mathbb{P}(T)$, $\mathbb{P}(V)$ et $\mathbb{P}(T)$. Soit la relation $r \in X \leftrightarrow Y$ de type $\mathbb{P}(T \times V)$. La *restriction sur le domaine* $E \lhd r$ est l'ensemble des couples $(x \mapsto y)$ de r pour lesquels x appartient à E. Son type est $\mathbb{P}(T \times V)$.

$$E \lhd r = \{x, y | x \mapsto y \in r \wedge x \in E\}$$

La *soustraction sur le domaine* $E \lhd\!\!\!- r$ est l'ensemble des couples $(x \mapsto y)$ de r pour lesquels x n'appartient pas à E. Son type est $\mathbb{P}(T \times V)$.

$$E \lhd\!\!\!- r = \{x, y | x \mapsto y \in r \wedge x \notin E\}$$

Définition 2.5 Restriction/soustraction sur le codomaine

Soit trois ensembles X, Y et F, de types respectifs $\mathbb{P}(T)$, $\mathbb{P}(V)$ et $\mathbb{P}(V)$. Soit la relation $r \in X \leftrightarrow Y$ de type $\mathbb{P}(T \times V)$. La *restriction sur le codomaine* $r \rhd F$ est l'ensemble des couples $(x \mapsto y)$ de r pour lesquels y appartient à F. Son type est $\mathbb{P}(T \times V)$.

$$r \rhd F = \{x, y | x \mapsto y \in r \wedge y \in F\}$$

La *soustraction sur le codomaine* $r \rhd\!\!\!- F$ est l'ensemble des couples $(x \mapsto y)$ de r pour lesquels y n'appartient pas à F. Son type est $\mathbb{P}(T \times V)$.

$$r \rhd\!\!\!- F = \{x, y | x \mapsto y \in r \wedge y \notin F\}$$

Définition 2.6 Surcharge par la droite

Soit deux ensembles X et Y, de types respectifs $\mathbb{P}(T)$ et $\mathbb{P}(V)$. Soit $r_1 \in X \leftrightarrow Y$ et $r_2 \in X \leftrightarrow Y$ deux relations de type $\mathbb{P}(T \times V)$. La *surcharge par la droite* (right overriding) $r_1 \Leftarrow r_2$ de r_2 sur r_1 est la relation constituée des éléments de r_2 et des éléments de r_1 dont le premier élément n'appartient pas au domaine de r_2. Son type est $\mathbb{P}(T \times V)$.

$$r_1 \Leftarrow r_2 = \{x, y | (x, y) \in X \times Y \wedge (((x \mapsto y) \in r_1 \wedge x \notin \mathrm{dom}(r_2)) \vee (x \mapsto y) \in r_2)\}$$

L'exemple suivant met en avant les restrictions sur les ensembles, en particulier le 'right overriding' pour mettre à jour – écraser – des éléments d'une relation.

Exemple 2.9. Soit l'ensemble énuméré $COUREURS = \{c_1, c_2, c_3, c_4, c_5\}$ qui représente l'ensemble des coureurs d'une course cycliste. Soit la fonction $place \in COUREURS \rightarrowtail\!\!\!\!\!\!+ \mathbb{N}$ qui donne la position d'arrivée des coureurs. *place* est en fait une fonction injective partielle : des coureurs différents auront des places d'arrivée différentes ; c'est une fonction partielle car au début de la course, aucun coureur n'a encore de place.

Supposons qu'à l'arrivée, les positions soient les suivantes, où 0 signifie une disqualification ou un abandon :

$$place = \{c_4 \mapsto 1, c_3 \mapsto 2, c_5 \mapsto 3, c_1 \mapsto 0, c_2 \mapsto 0\}$$

Un contrôle de dopage disqualifie le coureur c_4. On procède donc à la mise à jour :

$$place = place \mathbin{\lhd\!\!-} \{c_4 \mapsto 0, c_3 \mapsto 1, c_5 \mapsto 2\}$$

qui proclame c_3 vainqueur de la course, suivit de c_5.

2.3 Exemple

Dans les sections qui précédent, nous nous sommes attardés à décrire comment modéliser, à l'aide du langage B et de la machine abstraite, un problème. Nous avons ensuite vu quelles étaient les propriétés mathématiques à notre disposition pour renforcer ou développer notre solution.

Il est à présent temps de mettre en pratique toutes ces notions. Pour ce faire, nous nous penchons sur le problème de la traversée du pont[Rot02]. Nous rappelons dans un premier temps le problème, pour amener dans un second temps une solution en B classique.

2.3.1 Problème

Le problème s'énonce comme suit : quatre personnes se trouvent sur la rive gauche et souhaiteraient passer sur la rive droite en utilisant le pont. Malheureusement, la pénombre est tombée et, le pont étant fragile, seul 2 personnes peuvent se trouver sur celui-ci simultanément. Afin d'éclairer leur passage, une lampe de poche est mise à leur disposition. Elle doit toujours être portée lors de la traversée et ne peut être lancée d'une rive à l'autre, obligeant ainsi le retour d'une personne. Les temps de traversée des individus sont les suivants : 1 minute pour le premier, 2 minutes pour le second, 5 minutes pour le troisième et 8 minutes pour le dernier. Si 2 êtres traversent en même temps, ils doivent marcher à la vitesse du plus lent. Comment agencer l'ordre de passage de telle sorte que tout le monde se retrouve sur la rive droite en moins de 17 minutes ?

2.3.2 Solution

Pour résoudre le problème, nous avons construit une machine abstraite (voir le listing 2.2). Celle-ci est en fait une spécification du problème initial : nous avons modélisé, dans un formalisme précis qui est la méthode B, une solution à la traversée du pont. Cette solution regroupe un certain nombre de variables, d'états, et de transitions entre les états. Nous partirons de l'état initial (toutes les personnes se situent sur la rive gauche) pour ensuite modifier cet état par une série d'opérations (faire traverser des gens d'une rive à l'autre). Le but est d'atteindre un état final où tout le monde est à droite du pont, en un temps limité.

Listing 2.2 – Solution au problème de la traversée du pont

```
1  MACHINE Pont
2
3  SETS
4    LAMPE = {gauche, droite};
5    PERSONNES = {un, deux, trois, quatre}
6
7  CONSTANTS personnes, TEMPS
8
9  PROPERTIES
10   TEMPS ⊆ ℕ ∧ TEMPS = {1,2,5,8} ∧
11   personnes ∈ PERSONNES ⤖ TEMPS ∧
12   personnes = {un↦1, deux↦2, trois↦5, quatre↦8}
13
14 VARIABLES lampe, temps, riveg, rived
15
16 INVARIANT
17   lampe ∈ LAMPE ∧ temps ∈ ℕ ∧
18   riveg ⊆ PERSONNES ∧ rived ⊆ PERSONNES
19
20 INITIALISATION
21   lampe, temps, riveg, rived := gauche, 0, PERSONNES, {}
22
23 OPERATIONS
24
25   gauchedroite(p) = PRE p ⊆ riveg ∧ p≠{} ∧ card(p)≤2
       ∧ lampe=gauche
26                     THEN IF personnes[p]≠{}
27                          THEN lampe, temps, riveg, rived
                                :=
28                              droite, temps+ max(
                                    personnes[p]), riveg\p,
```

```
                                    rived∪p
29                              END
30                      END;
31
32    droitegauche(p) = PRE p ⊆ rived ∧ p≠{} ∧ card(p)≤2
          ∧ lampe=droite
33                      THEN IF personnes[p]≠{}
34                              THEN lampe,temps,riveg,rived
                                    :=
35                                  gauche,temps+ max(
                                        personnes[p]) ,riveg∪p,
                                        rived\p
36                              END
37                      END
38
39  END
```

Reprenons à présent la machine ligne par ligne et expliquons la en détail. Deux ensembles sont tout d'abord définis. `LAMPE` est l'ensemble des positions de la lampe de poche : elle se trouve soit à gauche, soit à droite de la rive. `PERSONNES` est l'ensemble des personnes devant traverser le pont : les éléments sont un, deux, trois et quatre.

Deux constantes sont ensuite définies. `TEMPS` est l'ensemble des temps de passages : 1, 2, 5 et 8. C'est un sous ensemble de l'ensemble des naturels. `personnes` est une fonction bijective – notée $\rightarrowtail\!\!\!\!\!\rightarrow$ dans le listing 2.2 – de `PERSONNES` vers `TEMPS` : pour chaque être humain, on donne son temps de passage du pont. Ces informations se retrouvent dans la clause `PROPERTIES`.

Quatre variables sont définies. La `lampe` qui donne le côté actuel où elle se trouve ; le `temps`, qui précise le temps actuel total de passage ; `riveg` et `rived` qui donnent les individus se trouvant respectivement sur la rive gauche et droite.

L'initialisation se déroule comme suit : la lampe de poche est à gauche, le temps total de passage est 0 et toutes les personnes sont sur la rive gauche.

Les deux opérations `gauchedroite` et `droitegauche` permettent de faire passer un ensemble de personnes – dénotés par la variable d'entrée p – d'une rive à l'autre. Ces opérations étant duales, nous ne détaillerons que la première (i.e. `gauchedroite`). Le paramètre d'entrée p est un sous ensemble de l'ensemble des personnes se trouvant à gauche de la rive. Il est composé d'1 ou 2 éléments.[12] L'opération change l'état des variables de la manière suivante : la lampe de poche est maintenant à droite, le temps est incrémenté du temps du maximum des personnes. Les compères traversant le pont sont enlevés des membres se trouvant sur la rive gauche (`riveg`) et ajoutés à ceux

[12] pour `gauchedroite`, il sera composé de 2 éléments, pour `droitegauche` de 1.

situés sur la rive droite (`rived`). Un test est ajouté pour vérifier que l'image `personnes[p]` de p est non vide. Ce test est nécessaire pour que `max(E)` soit défini[13].

2.3.3 Remarques

Le lecteur attentif aura remarqué que la solution apportée n'est pas complète : aucun ordonnancement n'est donné quand au passage des personnes d'une rive à l'autre. Il faudra attendre le chapitre 3 pour apporter la solution complète.

2.4 Non déterminisme

Dans tout ce qui précède, nous avons toujours travaillé avec une spécification *déterministe*, c'est-à-dire que l'exécution d'une opération permise dans n'importe quel état n'aboutit que dans un seul nouvel état. Alors que c'est une propriété essentielle pour la phase d'implémentation, elle n'est nullement requise par la spécification. Certains choix ne doivent en effet pas être pris en considération lors des premières étapes de modélisation, mais demandent plutôt notre attention lors de l'implémentation. Il est donc possible, pour une spécification, de trouver un état initial et des entrées données auxquels sont associés plus d'un état final et plusieurs sorties possibles : cette spécification est alors *non déterministe*.

Enonçons un petit problème dans lequel le non déterminisme est utilisé. Supposons un système de libération de places d'un restaurant, devant ajourner les personnes ayant fini leur repas. Il n'est pas nécessaire de préciser dans la modélisation d'un tel système quelle table va être libérée en premier[14]. Il est par contre indispensable de définir une opération qui fournit en sortie la table à mettre à la disposition des clients.

Différentes constructions en B classique font appel au non déterminisme. Nous n'aborderons ici que l'une d'entre elles : la construction **ANY**. En effet, celle-ci nous sera utile pour la suite du mémoire, notamment pour le chapitre 6.

2.4.1 Construction ANY

La construction ANY permet de choisir une ou plusieurs valeur(s) parmi un ensemble de variables qui satisfont toutes à une ou des conditions pour ensuite exécuter le corps de l'opération. Elle se construit comme suit :

```
1  ANY x
```

[13] Par définition, "E doit être non vide et doit posséder un majorant" [Cle02].

[14] par exemple, en supposant que les tables sont numérotées, sélectionner la table de numéro minimum et la libérer

```
2    WHERE P
3    THEN S
4    END
```

où x est une liste de variables, P est un prédicat qui porte sur les x et S sont des assignations. P doit également donner le type des variables x. Sa signification est la suivante : choisir une variable parmi x qui satisfait P et exécuter S (qui contient éventuellement la variable choisie). Ce choix est non déterministe.

Finalement, un exemple est apporté pour clarifier les idées.

Exemple 2.10. L'opération *nombre_impair* retourne un nombre impair compris entre 1 et 100. Ce nombre est complètement aléatoire.

```
1        resultat ← nombre_impair =
2        ANY x
3        WHERE x∈1..100∧(x mod 2) = 0
4        THEN resultat:=x
5        END
```

2.5 Raffinement

Le but de la méthode B est, rappelons-le, de modéliser un problème donné sous forme d'une machine respectant la notation AMN. Pour ce faire, une première phase de spécification est nécessaire. La première partie de ce chapitre était dédiée à cette tâche particulière. La deuxième partie, quant à elle, portera sur la phase suivante de modélisation qui est le *raffinement*. Cette phase est en fait une étape charnière entre d'une part la spécification qui reste à un niveau d'abstraction élevé et d'autre part l'implémentation qui est le code exécutable par un ordinateur.[15]

Notons dès à présent que la phase de raffinement n'est nullement obligatoire : il est donc possible de passer directement d'une spécification à une implémentation. Néanmois, dans bien des cas, cette phase est nécessaire car la spécification est d'un niveau tellement abstrait qu'une transcription directe engendrerait une perte de la vue d'ensemble du système.

Le raffinement est une succession d'étapes dont la dernière étape est l'implémentation. A chacune de ces étapes, nous décrivons de manière un peu plus détaillée *comment* seront représentées les variables d'état, *comment* les opérations seront exécutées. Le reste de ce chapitre a pour but de décrire ces étapes de raffinements. Nous devons auparavant introduire une série de constructions qui pourront être utilisées pour le raffinement.

[15] La phase d'implémentation ne sera pas abordée dans ce mémoire. Se référer aux documents *ad hoc*.

2.5.1 Constructions

Les constructions suivantes sont spécifiques au raffinement, c'est-à-dire qu'elle ne peuvent pas être utilisées pour la spécification. L'absence de ces dernières dans la spécification "empêche" l'utilisateur de penser en terme d'implémentation et encourage plutôt celui-ci à réfléchir à la modélisation de la solution d'un point de vue abstrait.

Composition séquentielle

Il est permis de définir des séquences pour le raffinement. Une séquence se définit comme $S;T$ où S et T sont des assignations et a comme sémantique : exécuter d'abord S et puis T. Un exemple suit.

Exemple 2.11. La variable x est incrémentée de 1, puis la variable y est décrémentée de x.

$$x := x + 1 \,;\, y := y - x$$

Si x est égal à 2 et y à 5, l'exécution de la séquence ci-dessus aura pour résultat que x vaut 3 et y vaut 2.

Variables locales

Il peut être nécessaire de devoir définir des variables locales, pour des états intermédiaires notamment. Ces variables locales sont définies à l'aide de la déclaration VAR. Plus généralement, une variable locale est définie comme VAR vl IN S END où *vl* est le nom de la variable locale et S est une (séquence d')assignation(s).

Exemple 2.12. Le swap des variables x et y de l'exemple 4.1 peut également s'écrire comme suit :

```
1  VAR t
2  IN
3  t := x ;
4  x := y ;
5  y := t ;
6  END
```

Définition de la machine

Un raffinement est une machine concrète définie en utilisant le mot-clé REFINEMENT, suivant du nom du raffinement.[16] Un raffinement s'effectue soit sur une machine abstraite, soit sur une machine concrète issue d'un autre raffinement ; il faut préciser quelle machine est raffinée : le mot-clé REFINES est là pour ça : il doit être suivi du nom de la machine.

[16] Une convention est de prendre le nom de la machine et d'y ajouter la lettre R.

Exemple 2.13. Pour raffiner la machine *Parking* de l'exemple 2.1, il faut procéder comme suit :

```
1  REFINEMENT ParkingR
2  REFINES Parking
```

Toutes les constructions qui ont été introduites pour les machines abstraites (variables abstraites, non-déterminisme, ...) sont toujours disponibles dans cette étape de modélisation.

2.5.2 Gluing invariants

Il est important de constater qu'un raffinement doit avoir exactement la même interface que la machine abstraite qu'elle implémente. En d'autres mots, elle doit offrir les mêmes opérations, avec les mêmes entrées et sorties. Cette restriction paraît à priori sévère mais il n'en est rien : un utilisateur externe ne doit *pas* pouvoir différencier une machine abstraite d'un raffinement.

Pour nous aider dans cette tâche, le B classique définit la notion de *gluing invariant*, qui lie machine abstraite et machine concrète.

Définition 2.7 Gluing invariant

L'invariant de la machine concrète *lie* les variables de la machine concrète aux variables de la machine abstraite. Cette relation est appelée *gluing invariant*.

Enonçons quelques propriétés liées au 'gluing invariant'.

Opérations

Chaque opération doit fournir exactement la même interface à la fois au niveau abstrait et au niveau concret. Si l'opération pour le raffinement fournit une valeur en sortie, alors l'opération de la spécification doit également fournir la même sortie. De plus, si l'opération abstraite est active, l'opération concrète doit également être active et préserver le 'gluing invariant'. Un raffinement est correct si les opérations préservent le 'gluing invariant'. De plus, le 'gluing invariant' doit être initialement vrai.

Les propriétés sur les opérations assurent que chaque séquence d'interactions sur la machine concrète est également possible pour la machine abstraite. Par conséquent, un utilisateur ne peut jamais savoir que la machine abstraite a été remplacée par une machine concrète.

Héritage et accès

Un raffinement hérite et a accès aux paramètres, ensembles et constantes de la machine qu'il raffine. Elles peuvent toutes apparaître soit dans l'inva-

riant, soit dans les opérations. L'invariant peut également faire référence aux variables de la machine abstraite.

2.5.3 Exemple

L'exemple que nous présentons a pour but de montrer comment se passe un raffinement en pratique. Cet exemple n'a aucune autre utilité et sa simplicité permet de se concentrer sur l'objectif. L'exemple est le suivant : on désire remplir une liste d'éléments, chaque élement étant un nombre naturel. La liste a une taille maximale de 300 éléments. La liste est ici un concept générique : nous raisonnons à un niveau d'abstraction élevé et ne devons donc pas détailler comment la liste est implémentée. Le listing 2.3 donne la machine abstraite de ce problème.

Listing 2.3 – Machine abstraite de l'exemple

```
1  MACHINE Exemple
2
3  VARIABLES liste
4
5  INVARIANT
6     liste ⊆ ℕ ∧ card(liste) ≤ 300
7
8  INITIALISATION
9     liste := {}
10
11 OPERATIONS
12
13    ajouter(el) = PRE el ∈ ℕ ∧ card(liste) < 300
14                  THEN liste := liste ∪ {el}
15                  END
16 END
```

Le raffinement correspondant se trouve au listing 2.4. Nous avons ici choisi de ne pas représenter les doublons dans la liste : c'est un choix d'implémentation qui a sa place dans cette phase de raffinement. Le nom de la machine est `ExempleR`, et la ligne 3 indique quelle machine nous raffinons (`Exemple`). La variable aa contient la liste d'éléments : c'est une fonction injective – pour éviter les doublons – du sous-ensemble des naturels allant de 1 à 300 vers les naturels. Par définition d'une fonction injective, on a $aa(i) = aa(i') \Rightarrow i = i' \quad \forall i \in \mathrm{dom}(aa)$. Le prédicat $ran(aa) = liste$ est le 'gluing invariant' : il permet de lier la liste abstraite $liste$ à la liste concrète aa (l'image de aa correspond à la liste abstraite). L'opération est également modifiée, pour prendre en compte la nouvelle définition de la liste. Remarquons que la signature est exactement la même : ce qui change, c'est le corps de l'opération. Le prédicat $el \notin \mathrm{ran}(aa)$ garantit que l'élément que l'on veut

insérer (*el*) ne se trouve pas encore dans la liste *aa*. Il est ici important de choisir `SELECT` au lieu de `PRE`, pour que la condition de l'opération dans le raffinement ne puisse pas être restreinte. Ensuite, une valeur naturelle comprise entre 1 et 300 et n'appartenant pas au domaine de *aa* est choisie en utilisant la constuction non-déterministe **ANY** ... **WHERE** ... **THEN**. Cette phase de raffinement n'est donc pas encore une implémentation car la machine n'est pas complètement déterministe : il faudra dès lors opérer d'autres raffinements ou tout du moins supprimer le non-déterminisme pour aboutir à une implémentation. L'assignation se passe comme suit : l'élément *el* est ajouté à la position x de la liste *aa*.

Listing 2.4 – Raffinement de l'exemple

```
1  REFINEMENT ExempleR
2
3  REFINES Exemple
4
5  VARIABLES aa
6
7  INVARIANT
8    aa ∈ 1..300 ⤔ ℕ ∧ ran(aa) = liste
9
10 INITIALISATION
11   aa := {}
12
13 OPERATIONS
14
15   ajouter(el) = SELECT el∈ℕ ∧ el∉ran(aa) THEN
16                   ANY x
17                   WHERE x ∈ 1..300 ∧ x ∉ dom(aa)
18                   THEN  aa := aa ∪ {x ↦ el}
19                   END
20                 END
21 END
```

Chapitre 3

Vérification, application et outils

L'objectif général de cette partie est, d'une part de modéliser une spécification d'un problème donné dans un formalisme particulier qui est ici la méthode B, et d'autre part de valider cette spécification. Les règles permettant de construire une telle spécification ont été abordées au chapitre précédent. Il convient à présent de s'attarder sur la deuxième étape : la vérification. Ce chapitre est divisé comme suit : tout d'abord, la notion de vérification et les règles associées sont explicitées ; ensuite, une application de ces règles au problème de la traversée du pont est effectuée ; finalement, un outil de vérification est présenté.

3.1 Vérification

L'idée d'une vérification est la suivante : nous sommes en possession d'une machine abstraite qui modélise un problème. De manière plus spécifique, nous avons défini des variables, un invariant, des contraintes, des opérations, ... Cependant, rien ne nous dit que la construction est valide : le modèle décrit-il bien le problème initial ? L'invariant – qui pose des conditions sur les variables – est-il toujours vrai quel que soit l'état dans lequel on se trouve ?

Pour répondre à ces questions, il faut définir des *obligations de preuve* : celles-ci définissent ce qu'il faut prouver pour valider le modèle élaboré. De manière informelle, elles s'énoncent comme suit :

1. la machine a du sens et est cohérente ;
2. il est possible d'affecter des valeurs aux paramètres ;
3. les ensembles différés et les constantes peuvent être instanciées ;
4. il existe des états initiaux qui satisfont l'invariant ;
5. l'initialisation établit l'invariant ;
6. les opérations préservent l'invariant.

La première obligation veut "simplement" dire que la construction décrit bien le problème dont il est question, et non un autre. Bien qu'elle soit une propriété souhaitée, elle s'avère en pratique difficile, voire impossible à prouver.

3.1.1 Précondition la plus faible

Avant d'expliciter les obligations de preuve de manière formelle, il convient de définir des notions utiles pour la suite.

La précondition est ce qu'il faut garantir avant l'exécution d'une opération. En d'autres mots, quelles sont les conditions qui doivent être satisfaites par les variables pour que l'exécution de l'opération soit permise. La postcondition est ce qu'il faut garantir après l'exécution d'une opération. La notion de *substitution* est importante pour les obligations de preuve : donnons-en une définition.

Définition 3.1 substitution

Soit x une variable et E une expression. Une substitution est représentée par

$$x := E$$

où l'expression E est dans un premier temps évaluée dans l'état inital et la valeur résultante est ensuite assignée à la variable x.

Il existe une précondition particulière appelée *précondition la plus faible* qu'il convient de définir.

Définition 3.2 Précondition la plus faible

Soit S une substitution, P un prédicat et e un état. On appelle *précondition la plus faible*, $[S]P$, un prédicat qui tient dans un état e si et seulement si chaque réalisation de S à partir de e termine dans un état dans lequel P tient. P est aussi appelé la *postcondition* de $[S]P$.

Illustrons celle-ci par un exemple.

Exemple 3.1. Soit x une variable entière. Pour s'assurer que $x < 10$ après l'exécution de $x := 0$, il suffit que $0 < 10$, ce qui est toujours vrai.

$$[x := 0](x < 10) = (0 < 10)$$

Un exemple moins trivial est le suivant, dans lequel x et y sont des variables. A chaque étape, la substitution la plus à droite est sélectionnée et

la nouvelle valeur remplace l'ancienne valeur dans le prédicat P associé.

$$\begin{aligned}
&[x := x - y; y := y + x; x := y - x](x = A \wedge y = B)\\
\equiv&[x := x - y; y := y + x](y - x = A \wedge y = B)\\
\equiv&[x := x - y](y + x - x = A \wedge y + x = B)\\
\equiv&(y = A \wedge y + x - y = B)\\
\equiv&(y = A \wedge x = B)
\end{aligned}$$

Pour que $x = A$ et $y = B$ après l'exécution, il faut que $y = A$ et $x = B$ avant l'exécution : l'opération opère donc un échange – swap – des variables x et y.

La grande force de la précondition la plus faible est qu'elle peut être dérivée de manière automatique par une série de règles.[1] Enonçons à présent celles-ci.

Règles

La notation utilisée ci-dessous pour les substitutions n'est qu'un sucre syntaxique : une notation équivalente ainsi qu'une description plus détaillée des règles est disponible à l'annexe B.

Dans ce qui suit, la notation $P[E/x]$ désigne le prédicat P où toutes les occurrences libres de x ont été remplacées par l'expression E. De façon analogue, $P[E, F/x, y]$ désigne le prédicat P où toutes les occurrences libres de x et y ont respectivement été remplacées par les expressions E et F.

Soit P, Q, R et S des prédicats ; x et y deux variables ; E et F deux expressions. Les règles qui établissent les préconditions les plus faibles sont les suivantes :

$$[skip]P = P \tag{3.1a}$$
$$[\textbf{BEGIN } S \textbf{ END}]P = [S]P \tag{3.1b}$$
$$[x := E]P = P[E/x] \tag{3.1c}$$
$$[x, y := E, F]P = P[E, F/x, y] \tag{3.1d}$$
$$[\textbf{CHOICE } S \textbf{ OR } R]P = [S]P \wedge [R]P \tag{3.1e}$$
$$[\textbf{PRE } Q \textbf{ THEN } S \textbf{ END}]P = Q \wedge [S]P \tag{3.1f}$$
$$[\textbf{IF } Q \textbf{ THEN } S \textbf{ ELSE } R]P = (Q \Rightarrow [S]P) \wedge (\neg Q \Rightarrow [R]P) \tag{3.1g}$$

3.1.2 Obligations de preuve

Il est à présent temps de formaliser les obligations de preuves, c'est-à-dire l'ensemble des preuves qui sont obligatoires pour garantir la validité de la machine abstraite issue de la modélisation. Nous nous aiderons pour

[1] Cette tâche sera de ce faite réalisée par des outils.

cela dans un premier temps de la notion de précondition la plus faible pour construire ces preuves, et dans un second temps des règles associées pour réduire les obligations.

Le tableau 3.1 fournit l'ensemble des obligations de preuve. Détaillons chacunes d'elles.

TAB. 3.1 – Obligations de preuve

Machine abstraite	**Obligation de preuve**
MACHINE $N(p)$	
CONSTRAINTS C	$\exists p \bullet C$
SETS St	
CONSTANTS k	
PROPERTIES B	$C \Rightarrow (\exists St, k \bullet B)$
VARIABLES v	
INVARIANT I	$B \wedge C \Rightarrow \exists v \bullet I$
INITIALISATION T	$B \wedge C \Rightarrow [T]I$
OPERATIONS $y \leftarrow op(x) =$ PRE P THEN S END ; ... END	$B \wedge C \wedge I \wedge P \Rightarrow [S]I$

La première concerne la clause `CONSTRAINTS` de la machine. Elle s'énonce comme suit : Soit C l'ensemble des contraintes et $x_1, x_2, ..., x_n$ l'ensemble des paramètres de la machine. Alors il existe des valeurs pour tous les paramètres qui rendent la contrainte vraie.

$$\exists x_1, x_2, ..., x_n \bullet C \tag{3.2}$$

Passons maintenant aux propriétés. Soit C l'ensemble des contraintes, St les ensembles, k l'ensemble des constantes et B l'ensemble des propriétés. Alors, quelles que soient les valeurs des paramètres, il existe des ensembles et des constantes qui satisfont les propriétés.

$$C \Rightarrow (\exists St, k \bullet B) \tag{3.3}$$

L'invariant à présent. Lorsque tous les paramètres ont été fixés, il doit exister des états valides de la machine, c'est-à-dire des états qui satisfont l'invariant.

$$B \wedge C \Rightarrow \exists v \bullet I \tag{3.4}$$

De manière analogue, lorsque tous les paramètres ont été fixés, l'initialisation T doit assigner des valeurs aux variables qui satisfont l'invariant I.

$$B \wedge C \Rightarrow [T]I \tag{3.5}$$

En d'autres mots, lorsqu'on amorce la machine, l'état initial doit être cohérent avec l'invariant.

Pour chaque opération de la machine, l'obligation de preuve suivante doit être prouvée : si la machine se trouve dans un état cohérent – où l'invariant est vrai – et que les préconditions de l'opération sont satisfaites, alors la machine se trouvera également dans un état cohérent après l'exécution de l'opération.

$$B \wedge C \wedge I \wedge P \Rightarrow [S]I \tag{3.6}$$

Pour les opérations qui produisent un effet de bord, la sortie n'entre pas en ligne de compte pour la consistance de la machine : il n'est donc pas indispensable de déclarer explicitement son type dans le corps de l'opération. Un contrôle de type est quand à lui effectué si diverses sorties peuvent être produites[2].

3.1.3 Preuves des raffinements

A l'instar des machines abstraites, les machines concrètes issues des raffinements doivent également être prouvées. Nous fournissons dès lors les obligations de preuves associées. Commençons par le corps des opérations. Dans ce qui suit, I et J sont les invariants de la machine abstraite et concrète respectivement. Soit C_1 l'ensemble des contraintes sur les paramètères et B_1 l'ensemble des propriétés de la machine abstraite. C_2 et B_2 sont définies de manière similaire pour le raffinement. L'obligation à prouver pour que l'opération concrète **PRE** P_1 **THEN** S_1 **END** raffine l'opération abstraite **PRE** P **THEN** S **END** est :

$$(C_1 \wedge C_2 \wedge B_1 \wedge B_2 \wedge I \wedge J \wedge P) \Rightarrow P_1 \tag{3.7}$$

$$(C_1 \wedge C_2 \wedge B_1 \wedge B_2 \wedge I \wedge J \wedge P) \Rightarrow [S_1]\neg([S](\neg(J))) \tag{3.8}$$

L'obligation (3.7) nous garantit que si la précondition abstraite est vraie, alors il doit en être de même pour la précondition concrète. A l'opposé, si la précondition abstraite est fausse, alors S_1 – le corps de l'opération concrète – peut "faire ce qu'il veut" : le résultat sera toujours un raffinement. L'obligation (3.8) est un peu plus compliquée à comprendre. Informellement, elle nous garantit que chaque fois que S_1 opère une transformation d'état, alors il est également possible pour S d'opérer une transformation d'un état lié et d'aboutir dans un nouvel état lui aussi lié. Une décomposition de l'obligation permet de mieux la comprendre.

[2] dans le cas d'un **IF** ... **THEN** ... **ELSE** par exemple

- $[S]\neg(J)$ est vrai si S est assuré d'établir $\neg(J)$
- $\neg[S]\neg(J)$ est vrai si certaines exécutions de S peuvent établir J
- $[S_1]\neg([S](\neg(J)))$ est vrai si n'importe quelle exécution de S_1 aboutit dans un état où certaines exécutions de S peuvent établir J

Dès lors, chaque exécution de S_1 peut être couplé avec S.

Dans le cas particulier où une opération produit une valeur en sortie, l'obligation doit quelque peu être modifiée. Etant donné que les opérations concrètes et abstraites ont la même signature, la même variable de sortie est présente dans la définition des deux opérations. Ces deux variables de sortie doivent être considérées séparément : il suffit d'effectuer un renommage de la variable concrète – disons y – en y'. Le corps de l'opération concrète (S_1) devient dès lors $S_1[y'/y]$. La condition nécessaire est que les sorties abstraites et concrètes doivent être identiques après l'exécution des opérations, c'est-à-dire $y = y'$. L'obligation de preuve (3.8) devient donc

$$(C_1 \wedge C_2 \wedge B_1 \wedge B_2 \wedge I \wedge J \wedge P) \Rightarrow [(S_1[y'/y])]\neg([S](\neg(J \wedge y = y'))) \quad (3.9)$$

Pour l'initialisation, soit T et T_1 les initialisations pour la machine abstraite et concrète respectivement. L'obligation (3.8) devient :

$$(C_1 \wedge C_2 \wedge B_1 \wedge B_2) \Rightarrow [T_1]\neg([T](\neg(J))) \quad (3.10)$$

3.2 Application

Afin de mieux comprendre les règles et obligations de preuve, nous les appliquons sur des exemples vus dans le chapitre 2. Le but est d'acquérir le plus de confiance dans la validité de notre solution. Cette section est divisée en deux parties : dans un premier temps, nous reprenons le problème de la traversée du pont et et la vérifions manuellement. Nous nous attardons ici à la preuve de l'initialisation (3.5) et des opérations (3.6). Dans un deuxième temps, nous reprenons l'exemple de raffinement du chapitre2 et prouvons les obligations (3.10) pour l'initialisation et (3.7) et (3.8) pour l'opération. Rappelons que ce processus – qui peut paraître long et fastidieux – ne sera pas réalisé en pratique mais plutôt délaissé aux outils adéquats.

3.2.1 Problème de la traversée du pont

La solution apportée au problème se trouve au listing 2.2 page 17.

Initialisation

L'obligation de preuve (3.5) s'écrit :

$$
\begin{array}{l}
TEMPS \subseteq \mathbb{N} \wedge TEMPS = \{1,2,5,8\} \wedge \\
personnes \in PERSONNES \rightarrowtail\!\!\!\!\!\rightarrow TEMPS \wedge \\
personnes = \{un \mapsto 1, deux \mapsto 2, trois \mapsto 5, quatre \mapsto 8\} \\
\Rightarrow [lampe, temps, riveg, rived := gauche, 0, PERSONNES, \{\}] \\
(lampe \in LAMPE \wedge temps \in \mathbb{N} \wedge riveg \subseteq PERSONNES \wedge \\
rived \subseteq PERSONNES)
\end{array}
$$

Après application de l'initialisation T, il reste à prouver que :

$$
\begin{array}{l}
(gauche \in LAMPE \wedge 0 \in \mathbb{N} \wedge \\
PERSONNES \subseteq PERSONNES \wedge \{\} \subseteq PERSONNES)
\end{array}
$$

Les éléments suivants apportent la vérité de l'obligation de preuve, où le premier cas est directement tiré de la définition de $LAMPE$:

1. $LAMPE = \{gauche, droite\} \Rightarrow gauche \in LAMPE$
2. $0 \in \mathbb{N}$;
3. un ensemble – ici $PERSONNES$ – est son propre sous-ensemble ;
4. l'ensemble vide est le sous-ensemble de tout autre – ici $PERSONNES$ – ensemble.

gauchedroite(p)

L'obligation de preuve (3.6) est ici :

$$
\begin{array}{l}
TEMPS \subseteq \mathbb{N} \wedge TEMPS = \{1,2,5,8\} \wedge \\
personnes \in PERSONNES \rightarrowtail\!\!\!\!\!\rightarrow TEMPS \wedge \\
personnes = \{un \mapsto 1, deux \mapsto 2, trois \mapsto 5, quatre \mapsto 8\} \wedge \\
lampe \in LAMPE \wedge temps \in \mathbb{N} \wedge riveg \subseteq PERSONNES \wedge \\
rived \subseteq PERSONNES \wedge p \subseteq riveg \wedge p \neq \{\} \wedge \mathrm{card}(p) \leq 2 \wedge \\
lampe = gauche \\
\Rightarrow (personnes[p] \neq \{\} \Rightarrow [lampe, temps, riveg, rived := droite, \\
temps + max(personnes[p]), riveg \setminus p, rived \cup p] \\
(lampe \in LAMPE \wedge temps \in \mathbb{N} \wedge riveg \subseteq PERSONNES \wedge \\
rived \subseteq PERSONNES)) \wedge \\
(personnes[p] = \{\} \Rightarrow (lampe \in LAMPE \wedge temps \in \mathbb{N} \wedge \\
riveg \subseteq PERSONNES \ \wedge rived \subseteq PERSONNES))
\end{array}
$$

Après application de la substitution des valeurs aux variables (la partie entre crochet contenant le symbole :=), la partie droite de l'implication ($[S]I$ dans l'obligation 3.6) se réduit à :

$$(personnes[p] \neq \{\} \Rightarrow (droite \in LAMPE \wedge$$
$$temps + max(personnes[p]) \in \mathbb{N} \wedge$$
$$riveg \setminus p \subseteq PERSONNES \wedge rived \cup p \subseteq PERSONNES))$$

L'obligation de preuve réduite aux éléments essentiels devient :

$$personnes \in PERSONNES \rightarrowtail\!\!\!\!\!\!\rightarrow TEMPS \wedge$$
$$personnes = \{un \mapsto 1, deux \mapsto 2, trois \mapsto 5, quatre \mapsto 8\} \wedge$$
$$temps \in \mathbb{N} \wedge riveg \subseteq PERSONNES \wedge$$
$$rived \subseteq PERSONNES \wedge p \subseteq riveg \wedge p \neq \{\} \wedge \text{card}(p) \leq 2 \wedge$$
$$personnes[p] \neq \{\}$$
$$\Rightarrow (droite \in LAMPE \wedge temps + max(personnes[p]) \in \mathbb{N} \wedge$$
$$riveg \setminus p \subseteq PERSONNES \wedge rived \cup p \subseteq PERSONNES)$$

que l'on peut prouver en considérant les éléments suivants :

1. $p \subseteq riveg \wedge p \neq \{\} \wedge \text{card}(p) \leq 2 \Rightarrow personnes[p] \neq \{\}$
2. $LAMPE = \{gauche, droite\} \Rightarrow droite \in LAMPE$
3. $temps \in \mathbb{N} \wedge max(personnes[p]) \in \mathbb{N} \Rightarrow temps + max(personnes[p]) \in \mathbb{N}$
4. $riveg \subseteq PERSONNES \Rightarrow riveg \setminus p \subseteq PERSONNES$
5. $rived \subseteq PERSONNES \Rightarrow rived \cup p \subseteq PERSONNES$

droitegauche(p)

L'obligation de preuve (3.6) est ici :

$$TEMPS \subseteq \mathbb{N} \wedge TEMPS = \{1, 2, 5, 8\} \wedge$$
$$personnes \in PERSONNES \rightarrowtail\!\!\!\!\!\!\rightarrow TEMPS \wedge$$
$$personnes = \{un \mapsto 1, deux \mapsto 2, trois \mapsto 5, quatre \mapsto 8\} \wedge$$
$$lampe \in LAMPE \wedge temps \in \mathbb{N} \wedge riveg \subseteq PERSONNES \wedge$$
$$rived \subseteq PERSONNES \wedge p \subseteq riveg \wedge p \neq \{\} \wedge \text{card}(p) \leq 2 \wedge$$
$$lampe = droite$$
$$\Rightarrow (personnes[p] \neq \{\} \Rightarrow [lampe, temps, riveg, rived := gauche,$$
$$temps + max(personnes[p]), riveg \cup p, rived \setminus p]$$
$$(lampe \in LAMPE \wedge temps \in \mathbb{N} \wedge riveg \subseteq PERSONNES \wedge$$
$$rived \subseteq PERSONNES)) \wedge$$
$$(personnes[p] = \{\} \Rightarrow (lampe \in LAMPE \wedge temps \in \mathbb{N} \wedge$$
$$riveg \subseteq PERSONNES \wedge rived \subseteq PERSONNES))$$

Après application de la substitution des valeurs aux variables (la partie entre crochet contenant le symbole :=), la partie droite de l'implication ($[S]I$ dans l'obligation 3.6) se réduit à :

$$
\begin{array}{l}
(personnes[p] \neq \{\} \Rightarrow (gauche \in LAMPE \wedge \\
temps + max(personnes[p]) \in \mathbb{N} \wedge \\
riveg \cup p \subseteq PERSONNES \wedge rived \setminus p \subseteq PERSONNES))
\end{array}
$$

L'obligation de preuve réduite aux éléments essentiels devient :

$$
\begin{array}{l}
personnes \in PERSONNES \rightarrowtail\!\!\!\!\!\rightarrow TEMPS \wedge \\
personnes = \{un \mapsto 1, deux \mapsto 2, trois \mapsto 5, quatre \mapsto 8\} \wedge \\
temps \in \mathbb{N} \wedge riveg \subseteq PERSONNES \wedge \\
rived \subseteq PERSONNES \wedge p \subseteq riveg \wedge p \neq \{\} \wedge \mathrm{card}(p) \leq 2 \wedge \\
personnes[p] \neq \{\} \\
\Rightarrow (gauche \in LAMPE \ \wedge temps + max(personnes[p]) \in \mathbb{N} \wedge \\
riveg \cup p \subseteq PERSONNES \wedge rived \setminus p \subseteq PERSONNES)
\end{array}
$$

que l'on peut prouver en considérant les éléments suivants :

1. $p \subseteq riveg \wedge p \neq \{\} \wedge \mathrm{card}(p) \leq 2 \Rightarrow personnes[p] \neq \{\}$
2. $LAMPE = \{gauche, droite\} \Rightarrow gauche \in LAMPE$
3. $temps \in \mathbb{N} \wedge max(personnes[p]) \in \mathbb{N} \Rightarrow temps + max(personnes[p]) \in \mathbb{N}$
4. $riveg \subseteq PERSONNES \Rightarrow riveg \cup p \subseteq PERSONNES$
5. $rived \subseteq PERSONNES \Rightarrow rived \setminus p \subseteq PERSONNES$

3.2.2 Raffinement

La solution apportée se trouve au listing 2.3 page 23 pour la machine abstraite et listing 2.4 page 24 pour le raffinement.

Initialisation

L'obligation de preuve (3.10) s'écrit :

$$
\begin{array}{l}
[aa := \{\}]\neg[liste := \{\}]\neg(aa \in 1..300 \rightarrowtail\!\!\!\!\!\rightarrow \mathbb{N} \wedge ran(aa) = liste) \\
= [aa := \{\}]\neg\neg(aa \in 1..300 \rightarrowtail\!\!\!\!\!\rightarrow \mathbb{N} \wedge ran(aa) = \{\}) \\
= (\{\} \in 1..300 \rightarrowtail\!\!\!\!\!\rightarrow \mathbb{N} \wedge ran(\{\}) = \{\}) \\
= true
\end{array}
$$

par définition d'une fonction (injective) partielle.

Opération ajouter(el)

L'obligation de preuve (3.7) est ici :

$$
\begin{aligned}
&liste \subseteq \mathbb{N} \wedge card(liste) \leq 300 \wedge aa \in 1..300 \rightarrowtail \mathbb{N} \wedge ran(aa) = liste \wedge el \in \mathbb{N} \\
&\wedge card(liste) < 300 \Rightarrow true \\
&= true
\end{aligned}
$$

L'obligation de preuve (3.8) est ici :

$$
\begin{aligned}
&liste \subseteq \mathbb{N} \wedge card(liste) \leq 300 \wedge aa \in 1..300 \rightarrowtail \mathbb{N} \wedge ran(aa) = liste \wedge el \in \mathbb{N} \\
&\wedge card(liste) < 300 \\
&\Rightarrow [\text{ANY } x \text{WHERE } x \in 1..300 \wedge x \notin dom(aa) \text{THEN } aa := aa \cup \{x \mapsto el\} \text{END}] \\
&\neg[liste := liste \cup \{el\}]\neg(aa \in 1..300 \rightarrowtail \mathbb{N} \wedge ran(aa) = liste)
\end{aligned}
$$

Après application de l'assignation de la valeur $liste \cup \{el\}$ à la variable $liste$, l'obligation se réduit à :

$$
\begin{aligned}
&liste \subseteq \mathbb{N} \wedge card(liste) \leq 300 \wedge aa \in 1..300 \rightarrowtail \mathbb{N} \wedge ran(aa) = liste \wedge el \in \mathbb{N} \\
&\wedge card(liste) < 300 \\
&\Rightarrow [\text{ANY } x \text{WHERE } x \in 1..300 \wedge x \notin dom(aa) \text{THEN } aa := aa \cup \{x \mapsto el\} \text{END}] \\
&\neg\neg(aa \in 1..300 \rightarrowtail \mathbb{N} \wedge ran(aa) = liste \cup \{el\})
\end{aligned}
$$

Après application de l'assignation $aa := aa \cup \{x \mapsto el\}$, l'obligation se réduit à :

$$
\begin{aligned}
&liste \subseteq \mathbb{N} \wedge card(liste) \leq 300 \wedge aa \in 1..300 \rightarrowtail \mathbb{N} \wedge ran(aa) = liste \wedge el \in \mathbb{N} \\
&\wedge card(liste) < 300 \\
&\Rightarrow (\forall x \cdot (x \in 1..300 \wedge x \notin dom(aa) \Rightarrow (aa \cup \{x \mapsto el\} \in 1..300 \rightarrowtail \mathbb{N} \\
&\wedge ran(aa \cup \{x \mapsto el\}) = liste \cup \{el\}))
\end{aligned}
$$

ce qui est vrai car

1. $x \notin dom(aa)$ et $el \in \mathbb{N}$ implique que $aa \cup \{x \mapsto el\} \in 1..300 \rightarrowtail \mathbb{N}$;
2. $ran(aa) = liste$ et $ran(\{x \mapsto el\}) = \{el\}$ implique que $ran(aa \cup \{x \mapsto el\}) = liste \cup \{el\})$.

3.3 ProB

Comme nous l'avons vu précédemment, la création et la validation des obligations de preuve est le point clé de la vérification d'une spécification B. Cette tâche, à priori manuelle, peut vite devenir longue et fastidieuse. Fort heureusement, des outils ont été conçus pour automatiser le processus et permettre ainsi à l'utilisateur de se concentrer sur d'autres tâches. Plusieurs

outils sont à disposition pour la méthode B : citons parmi les plus connus Atelier-B[Ste96], B-Toolkit[BCUL99] et ProB [LB03]. Nous présenterons principalement ProB, mais verrons également quelles sont les relations avec les autres outils.

ProB est un outil d'aide à la modélisation du langage B classique. Il est utilisé comme complément aux autres outils, à savoir Atelier-B et B-Toolkit. Ces derniers permettent en effet la vérification de la consistance de la machine[3]. Cette vérification se fait soit automatiquement, soit manuellement ; dans les 2 cas de figures, la génération des obligations de preuve est automatique. Lors d'une vérification manuelle, il peut être difficile d'investiger la preuve et/ou de trouver le problème : il peut en effet être demandé à l'utilisateur d'introduire une valeur particulière pour une variable, ou de rajouter une hypothèse pour compléter la preuve. Il sera plus approprié, dans ce cas-ci, d'utiliser un "détecteur" d'erreurs et générateurs de contre-exemples. ProB peut trouver ce type d'erreurs et en avertir l'utilisateur.

Dans ce qui suit, les 2 principales fonctionnalités de ProB – animateur et model checker – sont apportées, pour ensuite les appliquer sur un exemple concret.

3.3.1 Animateur

ProB peut dans un premier temps être utilisé comme animateur. L'animation de la machine s'opère étape par étape, en sélectionnant à chacune de ces étapes une opération à effectuer parmi celles disponibles dans l'état actuel. Un retour en arrière (i.e. retour dans l'état précédent l'exécution de la dernière opération) est éventuellement possible. Au début de l'animation, le choix est limité : il faut d'abord établir les éventuelles constantes et ensembles (une opération) et puis initialiser la machine (une opération). Par la suite, et suivant le nombre d'opérations de la machine et les préconditions de celles-ci, une liste de toutes les opérations permises dans l'état actuel est présentée : l'exécution d'une de celles-ci, au choix, entraînera le passage de l'état actuel dans un nouvel état où d'autres opérations – voire les mêmes si les conditions le permettent – sont disponibles.

La figure 3.1 présente l'interface générale de l'application : la fenêtre supérieure contient la machine abstraite ; la fenêtre inférieure est divisée en 3 parties : de gauche à droite, la description de l'état courant de la machine, une liste des opérations permises et l'historique ayant mené l'utilisateur jusqu'ici.

Un grand avantage de ProB en tant qu'animateur par rapport aux autres outils est le suivant :

> "[...] unlike the animator provided by the B-Toolkit, the user

[3] remarquons que ProB peut également, pour des ensembles finis, fournir cette vérification

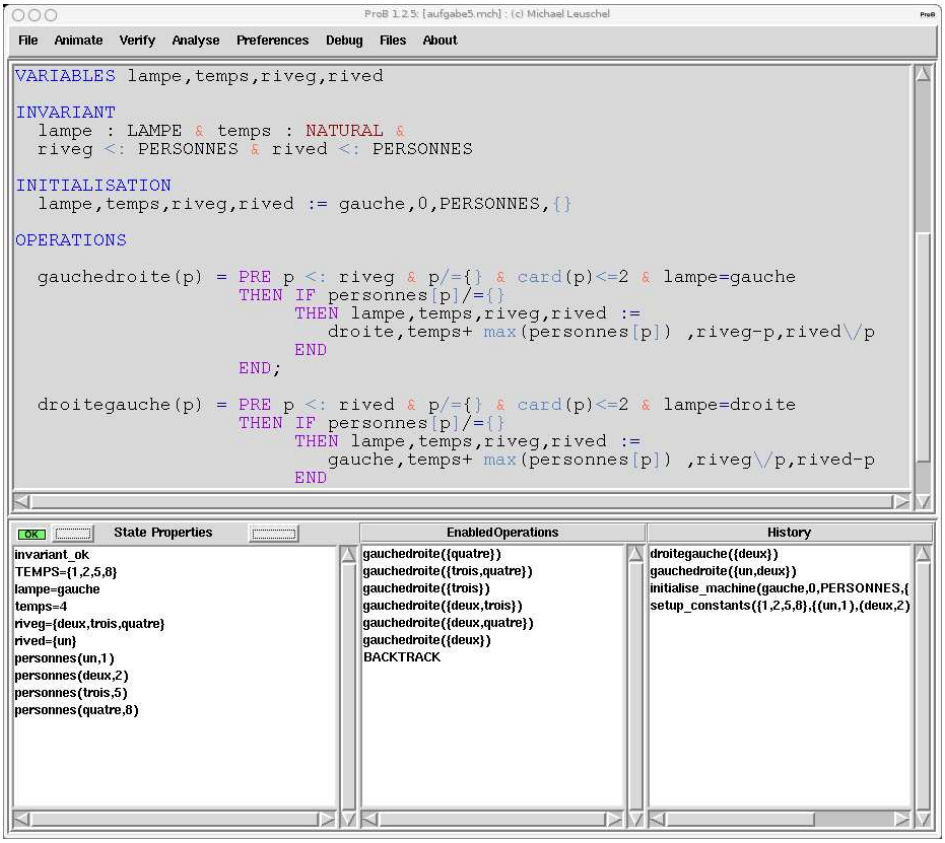

FIG. 3.1 – Animation dans PROB

> does *not* have to guess the right values for the operation arguments. The same holds for choice variables in `ANY` statements, the user does not have to find values which satisfy the `ANY` statement." [4] [LB03]

L'animateur autorise également la visualisation de l'espace d'état sous forme d'un graphe où les noeuds représentent les états et les transitions les opérations. Il est possible d'y voir l'état "courant", c'est-à-dire l'état où l'on se trouve à l'instant présent dans l'animation, le chemin nous ayant mené jusqu'ici et les états non encore explorés.

[4] traduction libre : "[...] à la différence de l'animateur fournit par le B-Toolkit, l'utilisateur ne doit pas estimer la valeur correcte pour les arguments de l'opération. La même chose tient pour le choix des variables dans les expressions `ANY`, l'utilisateur ne doit pas trouver des valeurs qui satisfont l'expression `ANY`."

3.3.2 Model checker

La deuxième fonctionnalité de ProB est le 'model checking'. Les 2 types de 'model checking' proposés sont :

1. *temporal model checking* : partant d'un état initial qui satisfait l'invariant, trouver une séquence d'opérations qui aboutissent dans un état qui viole l'invariant ;
2. *constraint-based checking* : trouver un état qui satisfait l'invariant et tel que l'exécution d'une seule opération dans cet état viole l'invariant.

Enonçons à présent les principales différences entre les 2 approches. Le 'temporal model checking' est plus simple à implémenter :

> "every single state is clearly determined, and we can use our ProB interpreter to compute all possible successor states of any given state, and then perform a search on the right sequence of operations." [LB03]

Le 'temporal model checking' est en général plus efficient, bien que le 'constraint-based checking s'avère utile dans certains cas, comme le souligne Mr. Leuschel : "This can be especially useful in circumstances where one has modified or added a single operation of a (previously verified) big machine." [LB03]

La difficulté du 'constraint-based-checking' vient du manque crucial d'informations de l'état à l'origine du viol d'invariant[5].

Pour le 'temporal model checking', le contre-exemple, accompagné de la trace ayant mené jusque-là sont disponibles pour l'utilisateur. Pour le 'constraint-based checking', l'état défectueux et la transition passant dans l'état erroné sont à disposition. Dans tous les cas, l'utilisateur hésitant est encouragé à tester les 2 approches.

3.3.3 Application

Pour cette application, nous reprenons l'exemple du problème de la traversée du pont[6]. L'animation réalisée ici est manuelle, c'est-à-dire que dans chaque état, l'utilisateur sélectionne l'opération à exécuter parmi celles disponibles. Il est également possible de réaliser une animation automatique, où l'on spécifie à ProB le nombre d'opérations à exécuter (par défaut 10) et il se charge du choix et de l'exécution de l'opération. Cette approche peut être utilisée pour exécuter un nombre important d'opérations. Il est alors aisé de retrouver le chemin parcouru en consultant l'historique ; le graphe est également à disposition pour une analyse plus poussée du tracé.

[5] voir [LB03] pour de plus amples informations ainsi que la solution apportée pour l'implémentation

[6] voir la sous-section 2.3.1 pour un rappel du problème

Procédons maintenant à l'animation. La figure 3.1 page 36 représente le status de ProB dans l'état où nous trouvons actuellement. La suite d'opérations effectuées se résume comme suit :

1. `setup_constants({1,2,5,8},{(un,1),(deux,2),(trois,5),(quatre,8)})`
2. `initialise_machine(gauche,0,PERSONNES,{})`
3. `gauchedroite({un,deux})`
4. `droitegauche({deux})`

Au début, nous n'avons pas le choix : il faut établir les constantes via l'opération prédéfinie *setup_constants*. Le premier argument de celle-ci établit la constante `TEMPS`, le second la constante `PERSONNES`. Afin d'initialiser la machine, nous devons exécuter, de nouveau sans alternative, l'opération *initialise_machine*. Celle-ci a pour but de placer la lampe à gauche, de mettre le temps à 0, et de placer toutes les personnes sur la rive gauche (et donc aucune sur la rive droite).

Nous arrivons dans un état où nous pouvons choisir une des opérations permises. Ce choix est totalement arbitraire, ProB se chargeant de la vérification des préconditions et de l'affichage des opérations permises. Notons que le nombre d'opérations peut être spécifié dans les préférences du programme : mettre une limite de 2 opérations ne signifie pas que seules 2 opérations seront permises mais bien que 2 seront affichées. Choisissons l'opération *gauchedroite({un,deux})*. Nous aboutissons dans un nouvel état où les opérations disponibles ne sont plus les mêmes que dans l'état précédent : nous avons en effet modifié la valeur de certaines variables d'état, comme le montre la description de l'état courant dans l'interface. Choisissons l'opération *gauchedroite({un,deux})*. De nouveau, un changement d'état est effectué.

La visualisation de l'espace d'état courant de la machine et du chemin jusque cet état est modélisé à la figure 3.2. A partir de l'état courant, il est toujours possible (sans dans l'état initial) de revenir à l'état précédent en "exécutant" `BACKTRACK` dans ProB : de là, il est éventuellement permis de choisir une autre opération (un autre arc sur le graphe) qui aboutira dans un nouvel état (le graphe est modifié en conséquence).

Utilisons à présent le 'model checking' pour trouver la solution au problème de la traversée du pont (i.e. faire traverser tout le monde en moins de 16 minutes). Pour cela, il faut substantiellement modifier la machine abstraite. Nous devons en effet trouver une manière de dire à ProB "trouve une suite d'opérations qui aboutissent dans un état où toutes les personnes sont à droite du pont et le temps est inférieur à 16 minutes". Il est possible de rajouter certaines fonctionnalités au 'temporal model checking' de ProB : trouver un 'deadlock', et/ou le viol de l'invariant, et/ou un but défini. C'est cette dernière option que nous choisissons. Un but est simplement un prédicat – que l'on spécifie par le mot clé `GOAL` dans la clause `DEFINITION` de

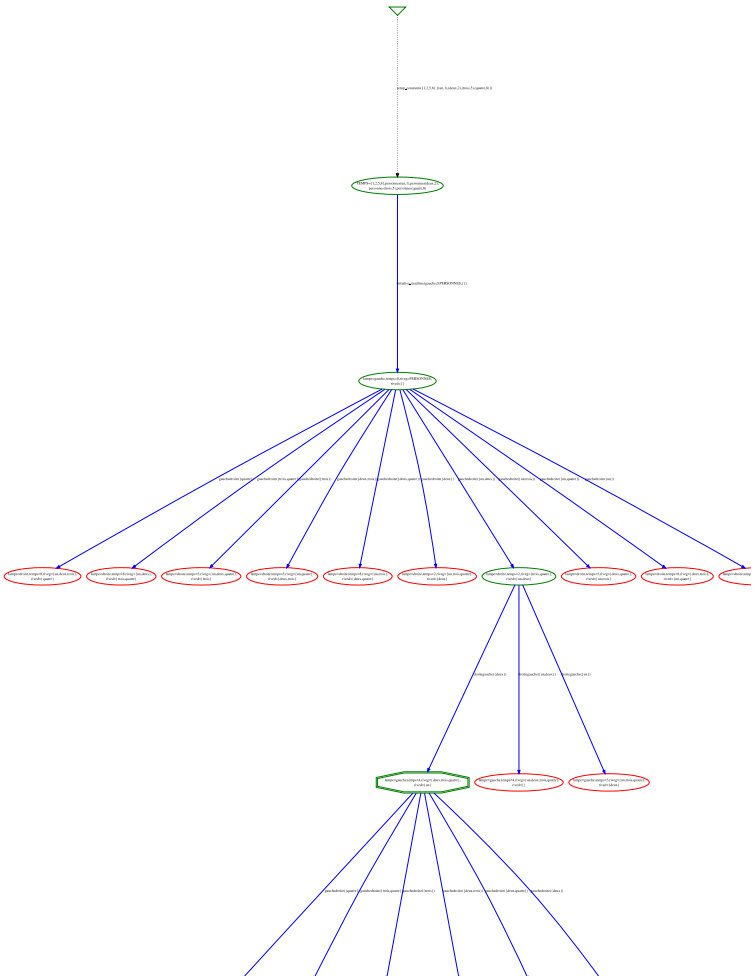

FIG. 3.2 – Visualisation de l'espace d'état

la machine – qui porte sur les variables. Typiquement, on veut que lorsque telle variable a telle valeur, où que telle autre valeur a atteint un certain seuil, on s'arrête. Dans notre cas, le but s'écrit

$$riveg = \{\} \wedge rived = PERSONNES \wedge TEMPS \leq 16$$

Une fois ce but encodé dans PROB, on utilise le 'model checking' en lui disant de trouver le but défini. Après quelques temps d'exécution, un état respectant le but est trouvé (en 15 minutes) et la suite d'opérations ayant mené à cet état est :

1. `setup_constants({1,2,5,8},{(un,1),(deux,2),(trois,5),(quatre,8)})`

2. `initialise_machine(gauche,0,PERSONNES,{})`
3. `gauchedroite({un,deux})`
4. `droitegauche({deux})`
5. `gauchedroite({trois,quatre})`
6. `droitegauche({un})`
7. `gauchedroite({un,deux})`

La solution en `ASCII` – c'est-à-dire la version écrite dans ProB– se trouve en annexe D. La conversion de la notation mathématique à la notation `ASCII` est assez triviale : nous ne nous attarderons donc pas sur celle ci[7].

Il reste finalement à tester le 'constraint-based checking'. Notre modèle étant bien construit, ProB ne trouve aucun état qui viole l'invariant.

[7] voir [Rob05] pour de plus amples informations

Deuxième partie

Le B événementiel

Chapitre 4

Introduction au B événementiel

Le B événementiel est considéré comme le successeur du B classique. C'est également une approche formelle à la spécification et au développement de systèmes informatiques. Il a notamment été introduit pour apporter des fonctionnalités manquantes telles que [Lec] les post conditions, les modalités, une syntaxe basée sur l'événement et un raffinement explicite. Comme dans le langage B, on retrouve les concepts de spécification, raffinements et implémentation. La notion de preuve est également importante, mais abordée sous un autre angle (se référer à ce sujet au chapitre 5).

Il est important de souligner que le B classique n'est pas un sous-ensemble du B événementiel : il est donc possible de modéliser certains problèmes en B classique, sans pour autant avoir une concordance parfaite avec le modèle correspondant en B événementiel. Il est tout de même envisageable, dans une certaine mesure, de passer d'un langage à l'autre.

A l'instar du chapitre 2, seul un sous-ensemble du langage sera abordé. Le lecteur désireux d'étendre ses connaissances peut se reporter à [ROD05b].

La structure du chapitre est la suivante : dans un premier temps, de nouveaux concepts mathématiques sont introduits, pour ensuite faire place à l'explicitation des deux concepts fondamentaux du B événementiel (modèle et contexte) ; finalement, un exemple permet d'expliciter les concepts et de comparer les deux langages introduits dans ce mémoire.

4.1 Concepts mathématiques

Avant de rentrer dans le vif du sujet, il convient de s'attarder sur certains nouveaux concepts mathématiques. Ceux-ci ont été introduits d'une part pour décrire le langage B événementiel d'un point de vue mathématique, et d'autre part pour utiliser ces concepts pour la preuve et la correction de systèmes. Ces concepts sont importants pour la compréhension du reste de

ce chapitre. Ils seront de même abondamment employés dans les chapitres suivants.

Les sous-sections qui suivent – de 4.1.1 à 4.1.3 – sont tirées d'une traduction libre des sections 2, 4 et 5 de la V^{e} partie de [ROD05b] intitulée "Event-B : Mathematical Language".

4.1.1 Prédicat et expression

L'objectif principal de cette sous-section est de définir de manière précise les concepts de *prédicat* et *expression*.

Prenons tout d'abord le temps de clarifier la distinction entre un prédicat et une expression. Un prédicat P est un fragment de texte formel qui peut être *prouvé* lorsqu'il fait partie d'un séquent[1], comme dans :

$$\vdash P$$

Un prédicat ne dénote pas quelque chose en particulier. Ce n'est pas le cas d'une expression qui dénote toujours un *objet*. Une expression ne peut pas être "prouvée". Par conséquent, les termes prédicat et expression sont incompatibles.

Avant d'énoncer les définitions formelles d'un prédicat et d'une expression, nous devons donner la syntaxe d'une variable et d'un ensemble. Une variable est dénotée simplement par un identificateur : sa syntaxe est $variable ::= identificateur$. La définition d'un ensemble est la suivante où var_list est une séquence finie de variables :

Définition 4.1 Ensemble

$$\begin{aligned} set ::= & set \times set && \text{(4.1a)}\\ & \mathbb{P}(set) && \text{(4.1b)}\\ & \{var\_list \cdot predicate | expression\} && \text{(4.1c)}\\ & variable && \text{(4.1d)} \end{aligned}$$

Soit s et t deux ensembles. La construction $s \times t$ est le produit cartésien de s et t (4.1a). La construction $\mathbb{P}(s)$ est le 'power set' de s (4.1b). Soit x une liste de variables d'identificateurs distincs, P un prédicat et E une liste d'expressions de la même taille que x. La construction $\{x \cdot P | E\}$ est le 'set defined comprehension' (4.1c).

Une expression se définit comme suit :

Une expression est donc soit une variable (4.2a), soit une expression de substitution (4.2b), soit un mapping (4.2c) d'une expression dans une autre, soit un ensemble (4.2d).

[1] voir à ce propos la sous-section 4.1.2

Définition 4.2 Expression

Une *expression* est une construction de la forme

$$\begin{aligned} expression ::= & variable && (4.2a)\\ & [var\_list := exp\_list]expression && (4.2b)\\ & expression \mapsto expression && (4.2c)\\ & set && (4.2d) \end{aligned}$$

où *exp_list* est une séquence finie d'expressions.

Poursuivons ensuite avec la définition d'un prédicat (4.3), dont on parle à la de la présente page.

Définition 4.3 prédicat

Un *prédicat* est une construction de la forme

$$\begin{aligned} predicate ::= & \top && (4.3a)\\ & \bot && (4.3b)\\ & \neg predicate && (4.3c)\\ & predicate \wedge predicate && (4.3d)\\ & predicate \vee predicate && (4.3e)\\ & predicate \Rightarrow predicate && (4.3f)\\ & predicate \Leftrightarrow predicate && (4.3g)\\ & \forall var\_list \cdot predicate && (4.3h)\\ & \exists var\_list \cdot predicate && (4.3i)\\ & [var\_list := exp\_list]predicate && (4.3j)\\ & expression = expression && (4.3k)\\ & expression \in set && (4.3l) \end{aligned}$$

Dans cette définition, $\top$ représente le prédicat vrai, tandis que $\bot$ représente le prédicat faux. (4.3c) est la négation d'un prédicat. (4.3d) et (4.3e) sont respectivement la conjonction et la disjonction de deux prédicats. (4.3f) et (4.3g) sont respectivement l'implication et l'équivalence de deux prédicats. Les constructions $\forall x \cdot P$ et $\exists x \cdot P$ sont respectivement appelées les prédicats quantifiés universellement (4.3h) et existentiellement (4.3i). La construction $[x := E]P$ est appelée le prédicat de substitution (4.3j). La construction $E = F$ est appelée le prédicat d'égalité (4.3k). La construction $E \in s$ est appelée le prédicat d'appartenance (4.3l).

4.1.2 Séquent et preuve

Comme pour le B classique, une propriété importante du langage repose sur la notion de *preuve*. Une fois la spécification modélisée dans le langage adéquat (ici le B événementiel), il est désirable de vouloir la prouver. Il faut pour cela définir ce qu'on entend par preuve, d'où l'utilité de cette sous-section. Nous verrons également une autre notion – le *séquent* – qui est fortement lié au concept de preuve.

Commençons par la définition formelle d'un séquent.

Définition 4.4 Séquent

Un séquent prend la forme suivante

$$H \vdash G$$

où H est un ensemble fini de prédicats appelés les *hypothèses*, et G est un prédicat appelé *but*

Ce séquent se lit comme suit : sous les hypothèses de l'ensemble H, prouver le but G. Il est à noter que H peut être vide.

Une *règle d'inférence* est une construction employée pour construire des preuves à partir de séquents. Elle se compose de deux parties : l'*antécédent* et la *conséquence*. L'antécédent dénote un ensemble fini de séquents – noté A – où chaque séquent est de la forme $H \vdash G$, alors qu'une conséquence – notée C – ne comprend qu'un séquent, également de la forme $H \vdash G$. Une règle d'inférence s'écrit de la façon suivante

$$r \quad \frac{A}{C}$$

Elle se lit comme suit : la règle r se réduit en une preuve du séquent C dès que l'on a des preuves pour chaque séquent de A. Dans le cas où l'antécédent A est vide, la règle d'inférence x s'écrit

$$x \quad \frac{}{C}$$

et se comprend : la règle x se réduit en une preuve du séquent C.

Une théorie $\mathcal{T}$ est un ensemble de règles d'inférence. La définition d'une preuve peut maintenant être formulée.

Par conséquence de la définition, les feuilles de l'arbre contiennent des règles sans antécédent. De plus, le noeud racine contient le séquent que nous désirons prouver.

Un exemple permet de clarifier les notions fraîchement introduites. Soit la théorie $\mathcal{T}$ suivante.

r1 $\frac{}{S2}$	**r2** $\frac{S7}{S4}$	**r3** $\frac{S2\ S3\ S4}{S1}$	**r4** $\frac{}{S5}$	**r5** $\frac{S5\ S6}{S3}$	**r6** $\frac{}{S6}$	**r7** $\frac{}{S7}$

Définition 4.5 Preuve

Soit $\mathcal{T}$ une théorie. La *preuve* d'un séquent dans $\mathcal{T}$ est un arbre fini composé de certaines contraintes. Les noeuds de l'arbre ont deux composantes : un séquent s et une règle r de la théorie $\mathcal{T}$. Les contraintes associées aux noeuds de l'arbre s'écrivent $s \mapsto r$: la conséquence de la règle r est s, et les fils de ce noeud sont les noeuds dont les séquents sont exactement tous les séquents de l'antécédent de la règle r.

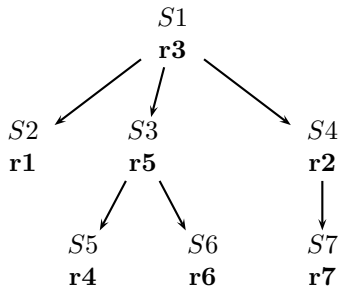

FIG. 4.1 – Preuve du séquent $S1$

La preuve du séquent $S1$ est donnée à la figure 4.1.

Comme nous pouvons le constater, la racine de l'arbre possède le séquent $S1$ qui est celui que nous voulons prouver. Il est facile de vérifier que chaque noeud, disons $S3 \mapsto \mathbf{r5}$, est tel que la conséquence de cette règle **r5** est bien le séquent $S3$. De plus, nous pouvons vérifier que les séquents des noeuds fils du noeud $S3 \mapsto \mathbf{r5}$, à savoir $S5$ et $S6$, sont exactement les séquents qui forment l'antécédent de la règle **r5**.

4.1.3 Théorie des ensembles et théorie de preuve

La théorie des ensembles, à l'instar des notations définies ci-dessus (prédicat, séquent, ...), constitue un élément de taille du langage mathématique introduit par le B événementiel. Ce langage mathématique a été construit de la sorte : à partir d'un sous-langage de base, ce dernier est progressivement étendu afin d'atteindre les objectifs explicités dans l'introduction de la section 4.1. Plus précisément, le but est de définir mathématiquement une théorie de preuve, définissant un ensemble de règles d'inférences sur les séquents. Cette théorie sera ensuite appliquée de la sorte : dans un premier temps, on définit une spécification en utilisant le langage B événementiel. Nous supposons ici que la spécification respecte autant la syntaxe que la sémantique du langage mathématique, c'est-à-dire qu'une phase de "compilation" nous aura prouvé la justesse syntaxique et sémantique de notre solution. La théorie de preuve reposant sur le langage mathématique d'une part, et notre spécification respectant ce même langage d'autre part, nous

sommes à même d'appliquer la théorie sur la spécification pour ainsi prouver notre modèle.

La propriété qui suit est un premier lien entre le B classique et le B événementiel. La génération de séquents et des preuves associées est un processus automatisable, tout comme l'était la génération d'obligations de preuve dans le B classique. Nous pouvons d'ailleurs remarquer que le terme *obligation de preuve* – ou Proof Obligation, PO – est un concept commun aux deux langages. Le chapitre 5 est notamment consacré aux obligations de preuve du B événementiel.

4.2 Modèles

Les deux concepts principaux du B événementiel sont le modèle et le contexte. Si nous désirons rapprocher la machine abstraite du B classique avec ces nouvelles notions de modèle et contexte, nous pouvons dire qu'une machine abstraite est l'union d'un modèle et d'un contexte. Les modèles et contextes permettent en effet d'isoler des parties spécifiques de la spécification, telles que les variables, invariants, constantes, ensembles, ... Nous exposons dans un premier temps les modèles ; les contextes sont quant à eux abordés dans la section 4.3.

4.2.1 Définition d'un modèle

De façon informelle, un modèle s'apparente à un automate (fini ou infini) muni de variables. Le modèle possède un état initial qui est caractérisé par les valeurs initiales des variables. A tout moment donné, l'état du modèle est caractérisé par les valeurs des variables. Il existe un ensemble de transitions – appelées ici *événements* – qui se produisent sous certaines conditions et qui permettent de passer d'un état dans un autre.

La définition 4.6 donne la composition d'un modèle.

Définition 4.6 Modèle

Un *modèle* se compose

1. d'un *nom*
2. d'une liste de *variables d'état* distinctes, dénotées par v ;
3. d'une liste de prédicats, les *invariants*, dénotés par $I(v)$;
4. d'un ensemble de transitions, appelées ici *événements*.

Chaque modèle porte un nom différent. L'invariant détermine, pour chaque variable du modèle, l'ensemble de valeurs que la variable peut prendre.

4.2.2 Evenements

Attardons-nous à présent au concept d'événement. Les événements correspondent en fait aux opérations du B classique. Un événement a pour but de modifier la valeur d'une ou plusieurs variables définies dans le modèle. Il doit d'abord satisfaire à des conditions, appelées ici *gardes* à la place de préconditions, avant de pouvoir être exécuté. Cette garde est en fait un prédicat construit à partir des constantes et des variables. Une fois les conditions requises – c'est à dire la garde satisfaite – le corps de l'événement, nommé *action*, peut être opéré. Ce corps correspond à l'assignation de nouvelles valeurs pour les variables, appelées ici *substitutions* et non plus *assignations* comme dans le B classique.

La définition 4.7 donne la composition d'un événement. La notion de substitution généralisée est explicitée à la sous-section 4.2.3.

Définition 4.7 Evénement

Un *événement* est composé des éléments suivants

1. un *nom* ;
2. un ensemble de prédicats, les *gardes*, dénotés par $G(v)$;
3. une *substitution* générale, dénotée par $S(v)$.

Comme pour le B classique, il existe un événement spécial, appelé *initialisation*, qui définit la situation initiale du modèle. Elle n'a pas de garde et a pour action d'assigner des valeurs à toutes les variables du modèle. Cette assignation s'effectue à l'aide des *substitutions* générales, que nous détaillons dans la sous-section qui suit.

4.2.3 Substitutions généralisées

Il existe 3 types de substitutions en B événementiel : la substitution multiple déterministe, la substitution vide, et la substitution multiple non déterministe.

La substitution multiple déterministe est le cas le plus général de substitution. Intuitivement, un sous-ensemble x de l'ensemble des variables v du modèle est sélectionné et de nouvelles valeurs sont assignées pour chaque variable apparaisant dans x. A l'extrême, il se peut que $x = v$, reflétant le cas où toutes les variables recoivent des nouvelles valeurs. Ce type de substition correspond à l'assignation multiple du B classique. Chaque substitution doit être écrite sur une ligne séparée et porte un nom (*act* 1 pour la 1re, *act* 2 pour la 2^{e}, ...). Donnons un petit exemple d'une telle substitution.

Exemple 4.1. L'échange – swap – des variables x et y s'écrit en B événementiel :

$$\begin{array}{ll} act\ 1 & x := y \\ act\ 2 & y := x \end{array}$$

La substitution vide est celle qui ne modifie rien. Elle s'écrit :

$$act\ 1 \quad \texttt{skip}$$

La substitution multiple non-déterministe correspond à la construction ANY du B classique (sous-section 2.4.1 page 19). Intuitivement, un choix non-déterministe permet de sélectionner, aléatoirement, une ou plusieurs variable(s) satisfaisant un ensemble de conditions et d'utiliser ces dernières dans le corps de l'événement. Sa syntaxe est la suivante :

any t **where**
  $P(t, v)$
then
  $x := F(t, v)$
end

où t dénote un ensemble de nouvelles variables distinctes de v et qui sont locales à la substitution ; $P(t, v)$ dénote une liste conjointe de prédicats et $F(t, v)$ est un nombre d'expressions de la théorie des ensembles qui correspondent à chacunes des variables de x, x étant un sous-ensemble de l'ensemble des variables v. Soulignons que ce type de construction est beaucoup plus utilisé dans ce langage. Les paramètres d'entrée sont en effet déclarés en utilisant cette construction, et n'apparaissent donc plus comme paramètre de l'opération comme c'était le cas en B classique. Fournissons un exemple pour expliciter cette dernière remarque.

Exemple 4.2. L'événement *incrementer* permet d'incrémenter la variable x d'une valeur entière positive i passée en paramètre.[2]

```
1  incrementer
2    ANY
3      i
4    WHERE
5      i ∈ ℕ1
6    THEN
7      x := x + i
8    END
```

4.3 Contextes

Comme précisé précédemment, les modèles apportent différents éléments intéressants dans l'élaboration d'une spécification : variables, invariants,

[2] Le numéro de la substitution *act* 1 n'apparait pas ici pour raison de clarté.

événements, substitutions, ... Cependant, certains concepts, tels que les constantes, les ensembles, ... qui sont disponibles dans le B classique n'ont pas encore été présentés dans ce chapitre. Ceux-ci font en fait partie du deuxième concept principal du B événementiel qui est le *contexte*.

Dans un premier temps, on donne la définition formelle d'un contexte pour ensuite expliciter la relation entre ces derniers et les modèles.

4.3.1 Définition d'un contexte

Commençons par donner la définition d'un contexte.

Définition 4.8 Contexte

Un *contexte* se compose

1. d'un nom ;
2. d'une liste de *carrier set*, notés par s ;
3. d'une liste de constantes distinctes, notées par c ;
4. d'une liste de propriétés nommées, notées par $P(s, c)$.

Un nom unique est toujours associé à un contexte. Ce nom est important car il sera utilisé pour lier ce contexte avec un ou plusieurs modèles (voir la sous-section 4.3.2 ci-dessous). Les *carrier sets* correspondent aux ensembles dans le B classique. Ceux-ci peuvent être énumérés (c'est à dire qu'on énumère la totalité de ses éléments) ou différés (seul le nom de l'ensemble est fourni, les éléments apparaissant dans l'initialisation ou les événements). Les constantes correspondent aux constantes dans le B classique, c'est-à-dire des variables qui ont une valeur fixe. Il est également permis de définir des propriétés sur ces constantes, celles-ci figurant dès lors dans les prédicats $P(s, c)$.

4.3.2 Lien entre modèles et contextes

Le lien entre modèle et contexte s'énonce comme suit : chaque modèle M peut référencer un contexte C. Dans ce cas, on dit que le modèle "voit" le contexte. Si le modèle M voit le contexte C, alors tous les *carrier sets* et les constantes définies dans C peuvent être utilisées dans M. La figure 4.2 montre la relation entre ces deux concepts.

Une spécification – et par extension un raffinement et une implémentation – doit obligatoirement posséder un modèle.Par contre, le contexte est facultatif. Un contexte est associé à un ou plusieurs modèles.

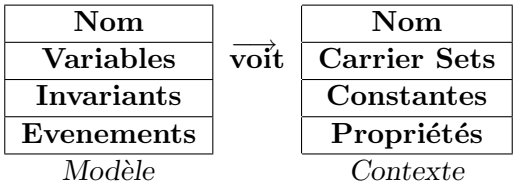

FIG. 4.2 – Lien entre modèle et contexte

4.4 Exemple

Ayant à présent vu la partie qui nous intéresse du B événementiel d'un point de vue théorique, il convient à présent de mettre en pratique les notions de modèles, contextes, ... Afin de pouvoir comparer les deux solutions, nous reprenons le même problème que précédemment, à savoir la traversée du pont[3] et le résolvons en utilisant le B événementiel. Ce problème, bien que relativement simple à résoudre, nous permettra tout de même de mettre en exergue les similitudes et différences des deux approches.

Au lieu de présenter la machine telle qu'elle apparaît dans le fichier source, nous préférons ici une représentation schématique de la solution. Cette dernière permet en effet de visualiser le lien entre modèle et contexte. Seules les opérations demanderont d'être détaillées.

4.4.1 Solution

La figure 4.3 apporte une vue schématique de la solution du problème. Cette dernière ressemblant très fort à la solution apportée pour le B classique, nous ne la détaillerons pas, mais soulignerons plutôt les différences entre les deux approches.

Le schéma est une construction classique du B événementiel dans lequel un modèle (**Pont**) "voit" un contexte (**PontContext**). Cela signifie, rappelons-le, que les *carrier sets* ($LAMPE$ et $PERSONNES$) et les constantes (*personnes* et $TEMPS$) sont mis à disposition du modèle : il n'est donc pas étonnant de voir ces derniers émerger dans l'invariant et les événements.

Il est intéressant de remarquer que les principales diversités entre les deux solutions se situent au niveau des opérations/événements, excepté pour l'initialisation. L'événement *droitegauche* étant "dual" à l'événement *gauchedroite*, nous ne détaillerons que le dernier. Le tableau 4.1 représente d'une part l'événement *gauchedroite*, d'autre part l'opération *gauchedroite*.

Cette représentation est utilisée afin de faciliter la comparaison entre les deux approches : il ne faut donc pas s'attarder sur la syntaxe globale d'un événement ou d'une opération mais plutôt les comparer ligne par ligne. De plus, la première colonne n'a de sens que pour la deuxième colonne : elle

[3] voir la sous-section 2.3.1 pour le rappel du problème

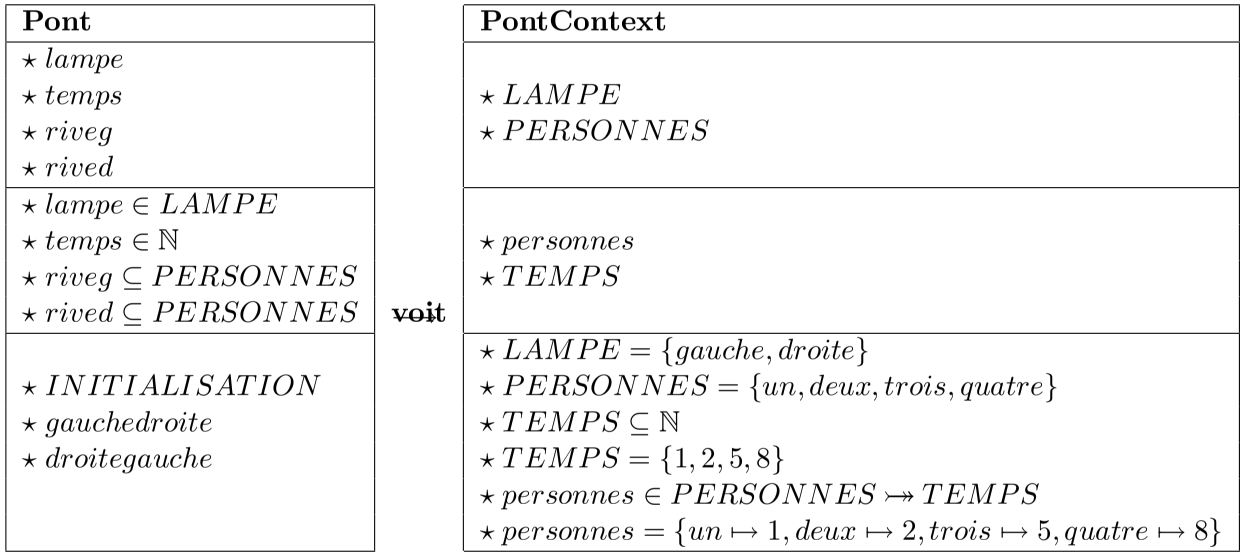

FIG. 4.3 – Schéma du modèle et du contexte associés au problème du pont

TAB. 4.1 – Evénement et opération *gauchedroite*

	Evénement	**Opération**
	p	p
$grd\ 1$	$p \subseteq riveg$	$p \subseteq riveg$
$grd\ 2$	$p \neq \varnothing$	$p \neq \{\}$
$grd\ 3$	$card(p) \leq 2$	$card(p) \leq 2$
$grd\ 4$	$lampe = gauche$	$lampe = gauche$
$grd\ 5$	$personnes[p] \neq \varnothing$	
		$IF\ personnes[p] \neq \{\}\ THEN$
$act\ 1$	$lampe := droite$	$lampe := droite$
$act\ 2$	$temps := temps+$ $max(personnes[p])$	$temps := temps+$ $max(personnes[p])$
$act\ 3$	$riveg := riveg \setminus p$	$riveg := riveg \setminus p$
$act\ 4$	$rived := rived \cup p$	$rived := rived \cup p$
		END
		END;

permet une numérotation des gardes et des actions, ceci à nouveau dans un but de comparaison simplifiée.

La différence principale qui apparaît dans le tableau concerne le prédicat $personnes[p] \neq \varnothing$[4]. Rappelons-le, ce prédicat est nécessaire car max n'est pas défini sur un ensemble vide. Pourquoi, dès lors, ne pas avoir employé une construction similaire au B classique – **IF** Q **THEN** S – mais plutôt l'avoir spécifié comme garde de l'événement ? Ce prédicat aurait également pu apparaître dans la précondition de l'opération *gauchedroite* mais nous avons préféré utiliser deux notations différentes pour faire ressortir la restriction suivante. Il est en effet impossible d'écrire directement en B événementiel une construction de type **IF** Q **THEN** S (**ELSE** R). Cette restriction peut toutefois être levée comme suit : dans le cas d'un **IF** Q **THEN** S, noter Q comme garde et S comme action ; dans le cas d'un **IF** Q **THEN** S **ELSE** R, créer les deux événements suivants :

1. une avec Q comme garde et S comme action ;
2. l'autre avec $\neg Q$ comme garde et R comme action.

4.5 Raffinement

Le texte ainsi que les figures qui suivent sont tirés d'une traduction libre de la section 3 de la 2e partie de [ROD05b] intitulée “Event-B : Structure

[4]Lorsque les deux syntaxes sont différentes, nous employons celle du B événementiel.

and Laws".

4.5.1 Raffinements de modèles et contextes

A l'instar du B classique, il est également possible d'effectuer des raffinements. Une spécification en B événementiel étant composée de modèles et de contextes, il va de soi que les raffinements concernent à la fois les modèles et les contextes. Clarifions le tout par une définition. Un exemple

Définition 4.9 raffinement
Soit M et N deux modèles tels que N a été construit à parti de M. Alors M est une *abstraction* de N et N est un *raffinement* de M. De manière similaire, soit C et D deux contextes tels que D a été construit à partir de C. Alors C est une *abstraction* de D et D est un *raffinement* de C.

d'une telle architecture utilisant des versions concrètes et abstraites de machines et contexes se trouve à la figure 4.4. Notons qu'il n'est pas nécessaire de raffiner le contexte C lors du raffinement d'un modèle M. Dans ce cas particulier, le modèle N voit le contexte C, tout comme le modèle M voit le contexte C. Cette situation est dépeinte à la figure 4.5

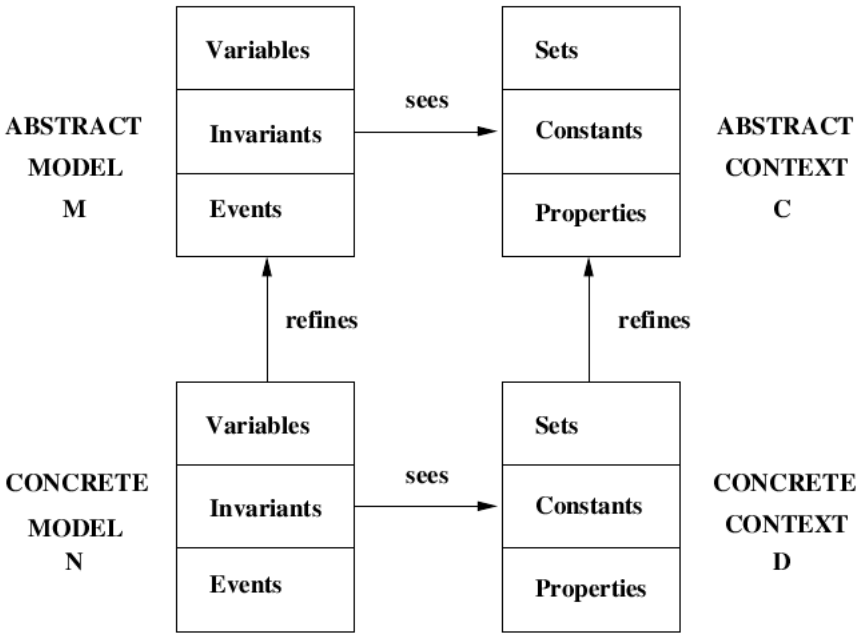

FIG. 4.4 – Raffinement d'un modèle et d'un contexte

Tous les ensembles et les constantes définis dans le contexte abstrait D sont conservés dans le raffinement. En d'autres mots, le raffinement d'un

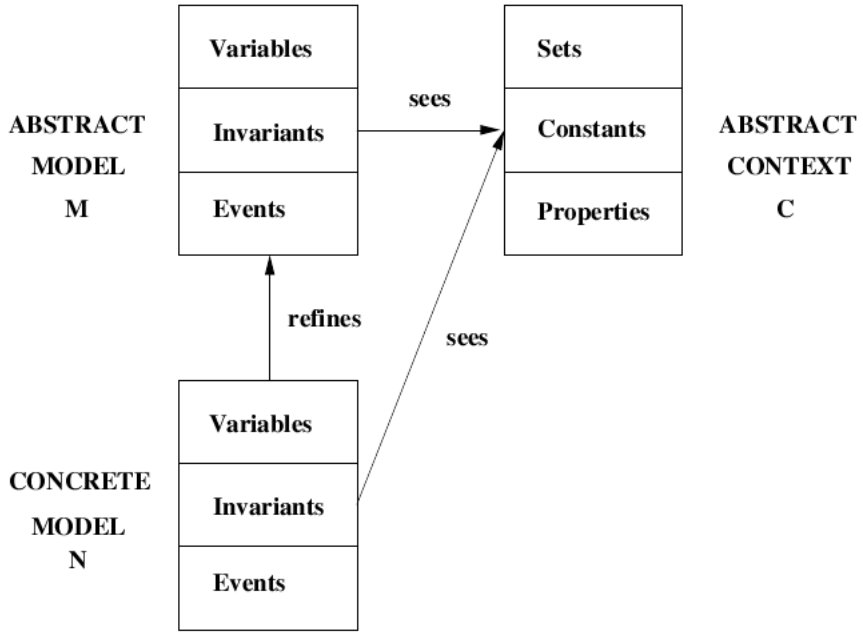

FIG. 4.5 – Cas particulier de raffinements d'un modèle et d'un contexte

contexte consiste simplement à ajouter des nouveaux *carrier sets* et des nouvelles constantes aux ensembles et constantes déjà existants. Dans la figure 4.4, le modèle N, qui voit le contexte D, peut dès lors utiliser tous les *carrier sets* et constantes définies dans D ainsi que dans C.

Pour les modèles, la situation est différente. Soit v l'ensemble des variables du modèle abstrait M et w l'ensemble des variables du modèle concret N. Alors les variables de w doivent être *complètement distinctes* des variables de v. De plus, contrairement à l'invariant de M qui dépend uniquement de v, l'invariant de N dépend de v *et* w. Comme pour le B classique, ce nouvel invariant porte le nom de '*gluing invariant*' et est dénoté par $J(s, c, v, w)$ où s et c sont les *carrier sets* et constantes *accumulées* – c'est-à-dire du contexte D et du contexte C – d'un raffinement. Sa signification est similaire à celle du B classique : il "lie" l'état du modèle concret N à celui du modèle abstrait M.

Le processus de raffinement peut être opéré à plusieurs niveaux comme le montre la figure 4.6. Notons tout de même que le 'gluing invariant' ne lie que deux modèles successifs. En d'autres mots, les variables présentes dans un 'gluing invariant' ne sont que celles du modèle correspondant et de son modèle abstrait.

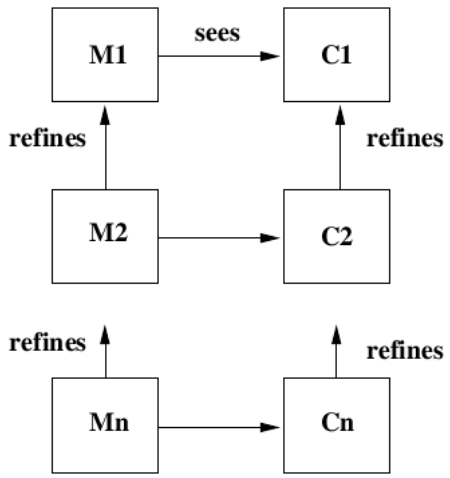

FIG. 4.6 – Raffinements de modèles et contextes

4.5.2 Raffinement d'événements existants

Le nouveau modèle N contient un nombre d'événements. Chaque événement qui se trouve dans la machine abstraite M *doit* être raffiné par un ou plusieurs événements dans la machine concrète N. Ce raffinement est illustré à la figure 4.7. Cela signifie que l'orsqu'un événement concret est défini, il doit dire explicitement quel événement il est supposé raffiner.

Notons qu'il existe d'autres propriétés pour le raffinement dans le B événementiel telles que l'introduction de nouveaux événements, la fusion d'événements abstraits, ... Nous ne les détaillons pas ici. Le lecteur curieux peut se reporter à la section 3.3 page 11 de la 2^{e} partie de [ROD05b] intitulée "Event-B : Structure and Laws".

4.5.3 Exemple

Nous reprenons l'exemple de raffinement de la sous-section 4.5.3 page 57 et l'appliquons au B événementiel. La solution en B événementiel s'apparantant très fort à la solution en B classique, nous ne nous attardons pas sur les modèles abstraits et concrets. Nous nous penchons toutefois sur les événements et utilisons le formalisme de la figure 4.7. Le schéma des événements abstraits et concrets des modèles `Exemple` et `ExempleR` se trouve à la figure 4.8.

Remarquons que bien que ces deux événements portent le même nom (`ajouter`), il faut préciser pour l'événement concret que l'on raffine l'événement abstrait. Cette précision n'avait pas lieu pour le B classique mais générera une erreur si elle est omise dans cette approche.

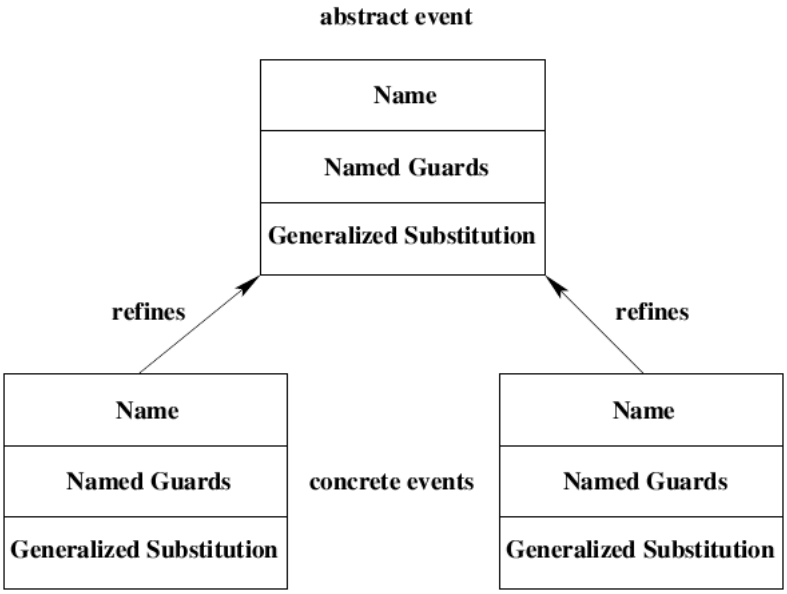

FIG. 4.7 – Un événement abstrait est raffiné par un ou plusieurs événements concrets

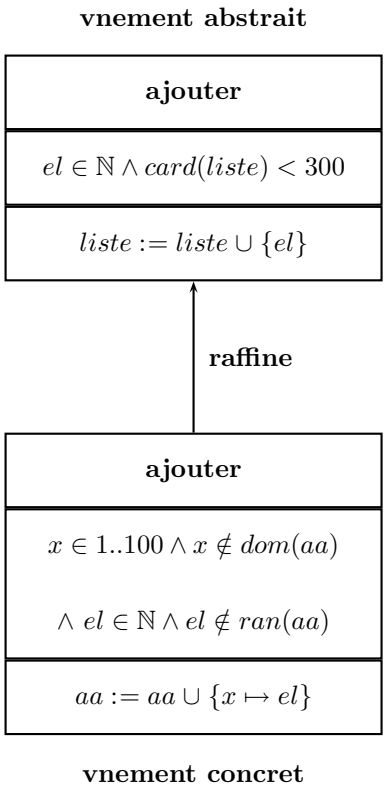

FIG. 4.8 – Schéma des événements abstraits et concrets de l'exemple

Chapitre 5

Vérification, application et outils

Le B événementiel, tout comme le B classique, s'utilise en deux étapes : tout d'abord une phase de modélisation permet d'aboutir à une spécification, un raffinement, ou une implémentation ; ensuite une phase de vérification permet de prouver la justesse de la modélisation. Dans ce chapitre, nous verrons comment les "objets" que l'on doit prouver, appelés *obligations de preuve*, sont générés et également comment il est possible de les prouver. L'exécution de cette tâche à la main devenant vite ingérable pour de grands projets, nous verrons également comment utiliser à bon escient le projet de recherche IST 511599 RODIN, dont l'objectif est de créer une méthodologie uniformisée et des outils de support pour le développement rigoureux et efficace d'un point de vue coût de systèmes informatiques. Des exemples et applications viendront également étayer l'exposé.

5.1 Génération des obligations de preuve

Les obligations de preuve pour le B événementiel sont quelque peu différentes de celles du langage B classique. Rappelons-le, les obligations sont générées en utilisant des règles associées à la notion de précondition la plus faible. L'extraction des obligations à partir d'une machine abstraite étant une tâche mécanique, elle est généralement effectuée par un outil adéquat. Les obligations de preuve pour le B événementiel sont générées à partir des modèles et des contextes, mais portent sur les événements (l'initialisation et ceux définis par l'utilisateur) des modèles. Cela peut s'expliquer comme suit : les contextes sont en quelque sorte des "outils" mis à la disposition des modèles : des *carrier sets*, des constantes, et un ensemble de propriétés. Un contexte isolé n'a aucune utilité en soit : c'est à partir du moment où il est mis en relation avec un modèle qu'il prend tout son sens.

Les sous-sections 5.1.1 et 5.1.2 sont tirées d'une traduction libre des

section 2 et 3 respectivement de la II^{e} partie de [ROD05b] intitulée "Event-B : Structure and Laws".

5.1.1 Obligations pour les modèles et les contextes

Nous verrons dans ce qui suit les trois obligations de preuves suivantes : faisabilité (**FIS**), préservation d'invariant (**INV**) et absence de deadlock (**DLKF**). Ces obligations portent sur les événements d'un modèle.

Dans ce qui suit, nous considérons les éléments suivants :

- M est un modèle avec v variables ;
- C est un contexte avec des *carrier sets* s et des constantes c ;
- le modèle M voit le contexte C ;
- les propriétés des constantes sont dénotées par $P(s, c)$, où s et c sont définis ci-dessus ;
- l'invariant est dénoté par $I(s, c, v)$;
- E est un événement de garde $G(s, c, v)$ et de prédicat $R(s, c, v, v')$, ce prédicat n'apparaissant que pour les substitutions multiples non-déterministes.

La première obligation s'énonce comme suit : étant donné les propriétés $P(s, c)$, l'invariant $I(s, c, v)$ et la garde $G(s, c, v)$, le prédicat R est garanti s'il existe au moins une valeur v' qui définit le prédicat $R(s, c, v, v')$. Cela veut dire qu'il existe bien des valeurs qui satisfont le prédicat d'une substitution multiple non-déterministe. La deuxième obligation exprime le maintien de l'invariant sous certaines conditions. En d'autres mots, si l'invariant est garanti avant l'exécution de l'événement, alors il est garanti après l'exécution de l'événement. Les obligations **FIS** et **INV** sont exprimées mathématiquement dans le tableau 5.1.

$P(s, c) \wedge I(s, c, v) \wedge G(s, c, v) \Rightarrow \exists v' \cdot R(s, c, v, v')$	**FIS**
$P(s, c) \wedge I(s, c, v) \wedge G(s, c, v) \wedge R(s, c, v, v') \Rightarrow I(s, c, v')$	**INV**

TAB. 5.1 – Faisabilité et préservation de l'invariant

Dans le cas de l'événement particulier *initialisation*, une simplification peut être opérée. Soit $RI(s, c, v\prime)$ le prédicat de la substitution multiple non-déterministe associé à un événement E. Les deux nouveaux éléments à prouver **INI_FIS** et **INI_INV** sont donnés dans le tableau 5.2.

$P(s, c) \Rightarrow \exists v' \cdot RI(s, c, v')$	**INI_FIS**
$P(s, c) \wedge I(s, c, v) \wedge RI(s, c, v\prime) \Rightarrow I(s, c, v')$	**INI_INV**

TAB. 5.2 – Faisabilité et préservation de l'invariant pour l'initialisation

Rappelons-le, il existe trois types de substitutions généralisées en B événementiel : la substitution multiple déterministe, la substitution vide – dénotée par `skip` – et la substitution multiple non déterministe. De manière

triviale, `skip` maintient l'invariant et n'a donc aucune obligation associée. Voyons à présent ce qu'il en est des deux autres possibilités en commençant par le cas déterministe. Une telle substitution s'écrit comme suit

$$\textbf{when } G(s,c,v) \textbf{ then } x := E(s,c,v) \textbf{ end}$$

La règle **FIS** associée est triviallement vraie et la règle pour l'invariant s'écrit

$P(s,c) \wedge I(s,c,v) \wedge G(s,c,v) \Rightarrow [x := E(s,c,v)]I(s,c,v')$	**INV_1**

TAB. 5.3 – Préservation de l'invariant pour une substitution déterministe

Une substitution non-déterministe s'écrit

when $G(s,c,v)$ **then**
  any t **where** $P(t,s,c,v)$ **then** $x := E(t,s,c,v)$ **end**
end

Les règles **FIS** et **INV** sont dans ce cas

$I(s,c,v) \wedge G(s,c,v) \Rightarrow \exists t \cdot P(t,s,c,v)$	**FIS_2**
$I(s,c,v) \wedge G(s,c,v) \wedge P(t,s,c,v) \Rightarrow [x := E(t,s,c,v)]I(s,c,v)$	**INV_2**

TAB. 5.4 – Faisabilité et préservation de l'invariant pour une substitution non-déterministe

Il existe également une possibilité de prouver l'absence de deadlock. Il suffit pour cela de dire que la disjonction des gardes des événements tient sous les propriétés des constantes et de l'invariant. Le tableau 5.5 donne cette règle, où $G_1(s,c,v), \ldots, G_n(s,c,v)$ dénotent les gardes de tous les événements.

$P(s,c) \wedge I(s,c,v) \Rightarrow G_1(s,c,v) \vee \ldots \vee G_n(s,c,v)$	**DLKF**

TAB. 5.5 – Absence de deadlock

Des exemples de générations d'obligations de preuve se trouvent à la sous-section 5.2.1 page 65.

5.1.2 Obligations pour les raffinements

Comme il a été explicité pour le B classique, il est de même possible – et préférable – de prouver les raffinements. Nous apportons dès lors les obligations de preuves associées. Soit un événement abstrait de garde $G(s,c,v)$ et de prédicat $R(s,c,v,v')$ et soit un événement concret de garde $H(s,c,w)$ et de prédicat $S(s,c,w,w')$ où v et w sont l'ensemble des variables des

$P(s,c) \wedge I(s,c,v) \wedge J(s,c,v,w) \wedge H(s,c,w)$ $\Rightarrow$ $\exists w' \cdot S(s,c,w,w')$	**FIS_REF**
$P(s,c) \wedge I(s,c,v) \wedge J(s,c,v,w) \wedge H(s,c,w)$ $\Rightarrow$ $G(s,c,v)$	**GRD_REF**
$P(s,c) \wedge I(s,c,v) \wedge J(s,c,v,w) \wedge H(s,c,w) \wedge S(s,c,w,w')$ $\Rightarrow$ $\exists v' \cdot (R(s,c,v,v') \wedge J(s,c,v',w'))$	**INV_REF**

Tab. 5.6 – Obligations de raffinement

modèles abstraits et concrets respectivement. Alors les obligations de preuve s'énoncent comme dans le tableau 5.6.

La première obligation **FIS_REF** exprime la faisabilité de l'événement raffiné. En d'autres mots, il est possible de trouver des valeurs pour l'ensemble des variables w' tel que le prédicat $S(s,c,w,w')$ est vrai. Les deuxièmes et troisièmes obligations, **GRD_REF** et **INV_REF** expriment le raffinement correct de l'événement raffiné par rapport à l'événement abstrait correspondant. Plus précisément, l'obligation **GRD_REF** nous certifie que si la garde $H(s,c,w)$ de l'événement concret est vraie, alors la garde $G(s,c,v)$ de l'événement abstrait est également vraie. L'obligation **INV_REF**, quant à elle, énonce que si l'événement concret est actif, alors il existe une valeur v' qui rend le prédicat $R(s,c,v,v')$ et le *gluing invariant* $J(s,c,v',w')$ après substitution des nouvelles valeurs pour v et w vrai.

Terminons cette sous-section par un cas spécial des événements. Soit les événements concrets et abstraits suivants :

when
  $G(v)$
then
  any t **where**
    $P(t,v)$
  then
    $v := E(t,v)$
  end
end

when
  $H(w)$
then
  any u **where**
    $Q(u,w)$
  then
    $w := F(u,w)$
  end
end

Pour question de simplicité, nous omettons dans la suite de mentionner les *carrier sets* s et les constantes c. Les trois obligations **FIS_REF**, **GRD_REF** et **INV_REF** se simplifient en[1] :

[1]Il existe en fait 4 cas spéciaux – d'où le chiffre 4 dans les obligations – mais nous ne présentons que celui qui nous intéresse.

$I(v) \land J(v,w) \land H(w)$ $\Rightarrow$ $\exists u \cdot Q(u,w)$	**FIS_REF_4**
$I(v) \land J(v,w) \land H(w)$ $\Rightarrow$ $G(v)$	**GRD_REF_4**
$I(v) \land J(v,w) \land H(w) \land Q(u,w)$ $\Rightarrow$ $\exists t \cdot (P(t,v) \land J(E(t,v), F(u,w)))$	**INV_REF_4**

TAB. 5.7 – Obligations de raffinement dans un cas spécial

5.2 Application

Cette section est divisée en deux parties : la première apporte les preuves des obligations du problème de la traversée du pont. Pour chaque événement (*initialisation*, *gauchedroite* et *droitegauche*), nous donnons les obligations de preuve ainsi que les preuves associées. La deuxième sous-section se rapporte à l'exemple du raffinement tel que décrit dans la sous-section 4.5.3. Nous nous attaquons à la preuve de l'événement *ajouter* dans la machine concrète `ExempleR`. Rappelons encore une fois que ce processus – qui peut paraître long et fastidieux – ne sera pas réalisé en pratique mais plutôt délaissé aux outils adéquats.

5.2.1 Problème de la traversée du pont

La solution apportée au problème se trouve au tableau 4.1 page 54.

Initialisation

L'initialisation ne contient pas d'obligation de faisabilité (**INI_FIS**). Seule une obligation de préservation de l'invariant (**INI_INV**) est présente : cette dernière porte sur la variable *temps* et stipule que sa valeur initiale – à savoir 0 – respecte l'invariant, c'est à dire

$$\vdash 0 \in \mathbb{N} \tag{5.1}$$

A priori, rien ne nous permet de prouver ce séquent car nous ne disposons d'aucune hypothèse. Cependant, les axiomes de Peano faisant partie intégrante du langage B événementiel, nous pouvons rajouter l'hypothèse $0 \in \mathbb{N}$ au séquent, ce qui conclut trivialement la preuve.

Evenement *gauchedroite*

La première obligation de preuve pour l'événement *gauchedroite* porte sur la faisabilité (**FIS**) du paramètre d'entrée p : définir $\text{card}(p) \leq 2$ revient

à prouver le séquent

$$p \neq \varnothing \wedge p \subseteq riveg \vdash finite(p) \tag{5.2}$$

Pour calculer la cardinalité d'un ensemble p, il faut que cet ensemble soit fini, d'où le prédicat $finite(p)$ dans le but du séquent. Un ensemble fini est dénombrable, ce qui peut se noter mathématiquement par $\exists f, n \cdot (n \in \mathbb{N} \wedge f \in 1 \ldots n \rightarrowtail\!\!\!\!\!\twoheadrightarrow s)$. En appliquant cette réécriture au séquent, il vient

$$p \neq \varnothing \wedge p \subseteq riveg \vdash \exists f, n \cdot (n \in \mathbb{N} \wedge f \in 1 \ldots n \rightarrowtail\!\!\!\!\!\twoheadrightarrow s) \tag{5.3}$$

Or, les définitions de plus petit et égal et de cardinalité permettent de dire que

$$\begin{aligned} a \leq b &\iff \exists c \cdot (c \in \mathbb{N} \wedge b = a + c) \\ n = \mathrm{card}(s) &\iff \exists f \cdot f \in 1 \ldots n \rightarrowtail\!\!\!\!\!\twoheadrightarrow s \end{aligned}$$

ce qui permet de déduire que $\exists f, n \cdot (n \in \mathbb{N} \wedge f \in 1 \ldots n \rightarrowtail\!\!\!\!\!\twoheadrightarrow s$

La deuxième obligation de preuve porte sur la préservation de l'invariant (**INV**) de l'événement. Elle s'énonce comme suit

$$\begin{aligned} personnes[p] \neq \varnothing \wedge p \neq \varnothing \wedge temps \in \mathbb{N} \wedge \mathrm{card}(p) \leq 2 \wedge lampe = gauche \\ \wedge\, p \subseteq riveg \vdash temps + \max(personnes[p]) \in \mathbb{N} \end{aligned} \tag{5.4}$$

La fonction max est définie sur les nombres naturels ; d'autre part, $personnes[p]$ est non vide et contient des éléments : on a donc que $\max(personnes[p]) \in \mathbb{N}$. L'addition de deux naturels $temps$ et $\max(personnes[p])$ est un nombre naturel, ce qui prouve le séquent.

La dernière obligation de preuve se subdivise en quatre obligations car il y a quatre actions dans l'événement *gauchedoite*. Nous ne nous attardons ici qu'à la deuxième action (*act* 2 dans le tableau 4.1 page 54), celle-ci étant la plus intéressante des quatre. Cette obligation est

$$\begin{aligned} personnes[p] \neq \varnothing \wedge p \neq \varnothing \wedge \mathrm{card}(p) \leq 2 \wedge lampe = gauche \\ \wedge\, p \subseteq riveg \vdash \exists b \cdot \forall x \cdot x \in personnes[p] \Rightarrow b \geq x \end{aligned} \tag{5.5}$$

Cette obligation de preuve ne peut être directement prouvée. Nous rajoutons dès lors de nouvelles hypothèses à la preuve : nous employons en fait des hypothèses avec des identificateurs libres et qui sont communs au but. Ces hypothèses sont les suivantes :

$$\begin{aligned} p \subseteq riveg \wedge riveg \subseteq PERSONNES \\ \wedge PERSONNES = \{un, deux, trois, quatre\} \\ \wedge personnes = \{un \mapsto 1, deux \mapsto 2, trois \mapsto 5, quatre \mapsto 8\} \end{aligned} \tag{5.6}$$

De ces hypothèses, nous pouvons conclure que $\forall x \cdot x \in personnes[p] \Rightarrow x \leq 8$ Nous avons dès lors trouvé un $b = 8$ qui permet de prouver le séquent ; ce b est en fait la valeur maximale que peut prendre $personnes[p]$ sous les diverses hypothèses citées ci-dessus. La fonction max telle que définie en *act* 2 prend donc bien sa valeur maximale pour 8.

Evénement *droitegauche*

La première obligation de preuve pour l'événement *droitegauche* est exactement la même que pour l'événement *gauchedroite* (5.2).

La deuxième obligation est fort semblable à l'obligation de preuve (5.4). La seule différence concerne la valeur de la variable *lampe* qui est tantôt à *gauche*, tantôt à *droite*. La valeur de cette variable n'ayant aucune importance pour la preuve du séquent, nous pouvons réutiliser la preuve de l'événement *gauchedroite* ; l'obligation est dès lors prouvée.

De manière analogue, la dernière obligation de preuve ressemble fort à l'obligation correspondante (5.5) : seule la valeur de la variable *lampe* diffère. Nous réutilisons donc une nouvelle fois une preuve déjà opérée : la dernière obligation est dès lors prouvée.

Absence de deadlock

Nous pourrions être tentés de tester l'absence de deadlock du système. Cependant, ce test serait inutile car nous ne désirons pas obtenir un système qui ne bloque pas, mais plutôt un système qui aboutit à un moment donné à une solution finale. Autrement dit, nous cherchons à atteindre un état final dans lequel toutes les personnes sont à droite de la rive et le temps est inférieur à 16 minutes.

5.2.2 Raffinement

La solution apportée se trouve à la figure 4.8 page 59.

Evénement *ajouter*

La première obligation de preuve (**FIS_REF_4**) se rapporte à la faisabilité de l'événement raffiné. Elle s'énonce comme suit :

$$\begin{aligned} liste \subseteq \mathbb{N} \land card(liste) \leq 300 \land aa \in 1..300 \rightarrowtail \mathbb{N} \land ran(aa) = liste \land true \\ \Rightarrow \exists x, el \cdot x \in 1..300 \land el \in \mathbb{N} \land x \notin \mathrm{dom}(aa) \land el \notin \mathrm{ran}(aa) \end{aligned} \tag{5.7}$$

L'obligation est fausse dans le cas où la liste *aa* est remplie, c'est-à-dire quand elle contient 300 éléments. Il est dès lors normal de ne pas pouvoir ajouter un élément à la liste. Par contre, si la liste n'est pas remplie, c'est-à-dire quand $card(aa) < 300$, alors il est possible de trouver une valeur pour

x telle que $x \notin \mathrm{dom}(aa)$. De plus, l'ensemble des nombres naturels étant infini, il existe une valeur naturelle qui n'appartient pas au 'range' de aa.

La deuxième obligation (**GRD_REF_4**) se prouve trivialement. L'événement *ajouter* dans le modèle comme dans le raffinement est dépourvu de garde, et on peut donc écrire $H(w) = true$ et $G(v) = true$. L'obligation se réduit donc à

$$true \Rightarrow true$$

qui est vraie.

La troisième obligation (**INV_REF_4**) s'écrit :

$$\begin{aligned} liste \subseteq \mathbb{N} \wedge card(liste) \leq 300 \wedge aa \in 1..300 \rightarrowtail\!\!\!\!\!\rightarrow \mathbb{N} \wedge ran(aa) = liste \wedge true \\ \wedge\ x \in 1..300 \wedge el \in \mathbb{N} \wedge x \notin \mathrm{dom}(aa) \wedge el \notin ran(aa) \\ \Rightarrow \exists x, el \cdot (el \in \mathbb{N} \wedge card(liste) < 300 \wedge\ aa \cup \{x \mapsto el\} \in 1..300 \rightarrowtail\!\!\!\!\!\rightarrow \mathbb{N} \\ \wedge\ ran(aa \cup \{x \mapsto el\}) = liste \cup \{el\}) \end{aligned} \tag{5.8}$$

qui est vraie car

1. $el \in \mathbb{N}$ implique que $el \in \mathbb{N}$;
2. $x \in 1..300 \wedge x \notin \mathrm{dom}(aa) \wedge aa \in 1.300 \rightarrowtail\!\!\!\!\!\rightarrow \mathbb{N} \wedge \mathrm{ran}(aa) = liste$ implique que $card(liste) < 300$;
3. $x \notin dom(aa)$ et $el \in \mathbb{N}$ implique que $aa \cup \{x \mapsto el\} \in 1..300 \rightarrowtail\!\!\!\!\!\rightarrow \mathbb{N}$;
4. $ran(aa) = liste$ et $ran(\{x \mapsto el\}) = \{el\}$ implique que $ran(aa \cup \{x \mapsto el\}) = liste \cup \{el\}))$.

5.3 RODIN

RODIN est le projet de recherche IST 511599 fondé par l'Union Européenne et dont le but est de développer une plateforme d'outils basés sur Eclipse et qui supporte le B événementiel. L'objectif de ce projet est de créer une méthodologie uniformisée et des outils de support pour le développement rigoureux et efficace d'un point de vue coût des systèmes informatiques.

Dans ce qui suit, nous verrons l'architecture globale de la plateforme, ses différents éléments et les liens qui existent entre eux. Nous verrons également comment a été modélisée la solution au problème de la traversée du pont grâce à cette plateforme d'outils, et comment cette solution a pu être prouvée. Les figures, définitions et une partie du contenu des Sous-sections 5.3.1 à 5.3.3 sont tirées de la partie "The Architecture of the Rodin Plateform" de [ROD05a] intitulée "Specification of Basic Tools and Plateform".

5.3.1 Architecture de la plateforme

La plateforme RODIN est découpée en trois parties :

– la plateforme Eclipse ;
– le kernel (ou core) plugins ;
– les plugins externes

La figure 5.1 est une représentation plus détaillée de cette découpe.

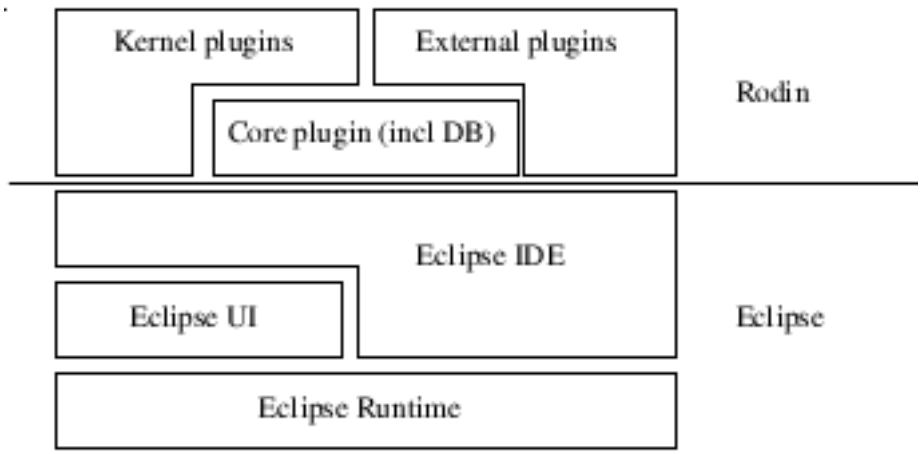

FIG. 5.1 – Architecture de la plateforme RODIN

La plateforme Eclipse fournit les outils basiques pour la construction d'une IDE (Integrated Development Environment). Dans RODIN, Eclipse a été utilisé pour supporter le processus de modélisation et de preuve du B événementiel. Les services de bases fournis par la plateforme, notamment l'architecture des plugins, la notion de workspace, l'interface graphique ont été réutilisés. Toute bonne documentation d'Eclipse fournit une explication détaillée des divers services de bases.

Le *kernel plugin* fournit les facilités basiques afin de modéliser du B événementiel. Il peut être décomposé en un ensemble de plugins qui sont "pluggées" dans la plateforme Eclipse :

– **Core** : il fournit d'une part les routines générales et d'autre part un gestionnaire de base de données, ce dernier enregistrant tout ce qui est relié aux modèles du B événementiel, notamment les obligations de preuve et les preuves.
– **Static Checker** : il contient les routines nécessaires pour vérifier que le contenu de la base de données est significatif pour le B événementiel.
– **Proof Obligation Generator** : il génère les obligations de preuve pour les modèles et les contextes.
– **Prover** : il fournit un *prover* mécanique utilisé pour *décharger* les obligations de preuve.
– **Modelling UI** : il fournit l'interface graphique pour écrire des modèles en B événementiel.
– **Proof UI** : il fournit l'interface graphique pour *décharger* des obliga-

tions de preuve.

Finalement, les plugins externes sont tous les autres plugins qui peuvent être utilisés dans la plateforme RODIN. Le chapitre 6 traite notamment de l'un de ces plugins externes.

5.3.2 Utilisation

L'utilisation de RODIN se divise en deux étapes principales : tout d'abord la construction d'un modèle, puis ensuite la preuve de la justesse de ce modèle. Pour chaque étape, une perspective est accessible dans l'interface de RODIN, comme le montre la figure 5.2 de la présente page pour la perspective *Event B* et la figure 5.3 page ci-contre pour la perspective *Proving*.

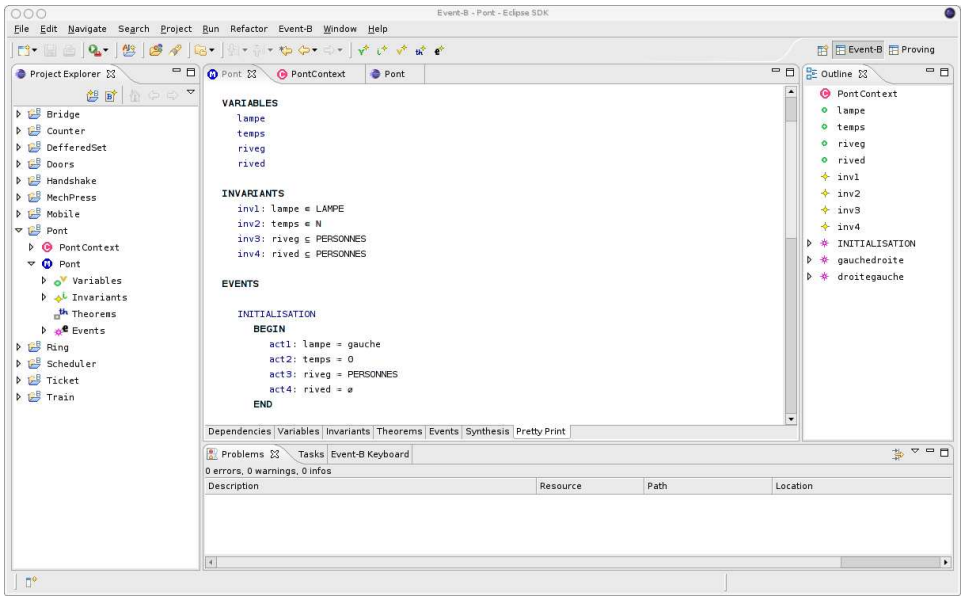

FIG. 5.2 – Capture d'écran de la perspective *Event-B* dans RODIN

La première étape est la construction du modèle et du contexte à proprement parler. L'interface propose quelques facilités pour cette tâche, telles que les 'wizard' qui permettent d'introduire variables, invariants, événements, ... plus rapidement.[2] La vue 'Pretty Print' permet à la fois pour les modèles et pour les contextes d'obtenir une vue globale du système.

L'étape de preuve se subdivise en deux tâches : une dérivation des obligations de preuve associées au modèle, et le *déchargement* de ces dernières grâce à un *prover* mécanique. La grande majorité de ces tâches sont – comme nous l'avons vu dans la section 5.2 – fastidieuses et peuvent être automatisées

[2]Pour une variable, par exemple, on introduit le nom (*lampe*), et les champs initialisation et invariant sont directement complétés par $lampe :=$ et $lampe \in$

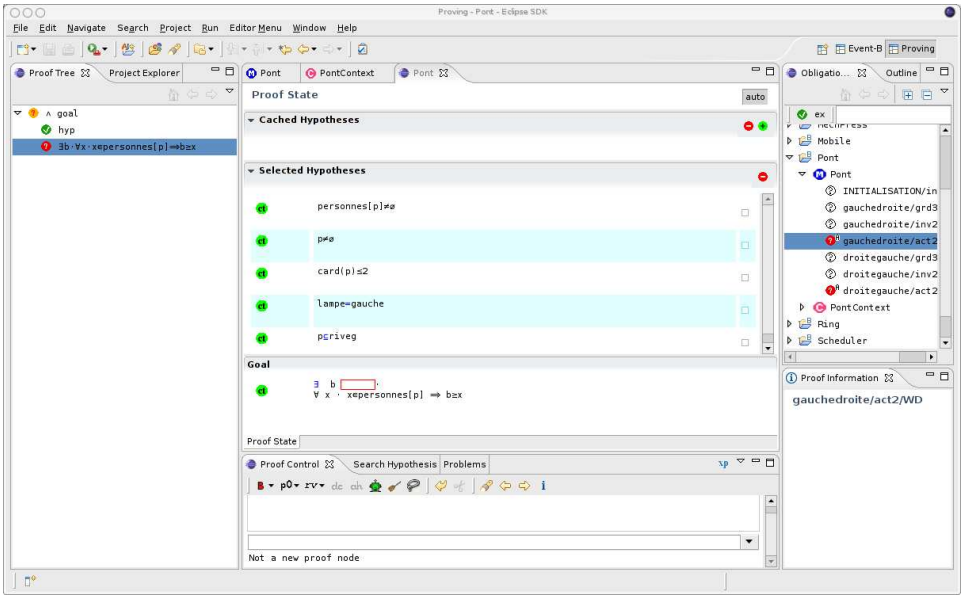

FIG. 5.3 – Capture d'écran de la perspective *Proving* dans RODIN

à l'aide d'outils adéquats. De plus, pour atteindre l'objectif défini ci-dessus, nous souhaitons que ce travail soit presqu'entièrement caché. L'utilisateur ne doit voir que son modèle, et les obligations de preuve *intéressantes* qui sont dérivées du modèle, c'est-à-dire les obligations de preuve qui ne peuvent pas être *déchargées* automatiquement.

L'architecture de construction se trouve à la figure 5.4. Cette architecture modélise les différentes phases et les acteurs qui entrent en jeu dans le processus de modélisation et de preuve d'un modèle. Les rectangles en plusieurs couches représentent les fichiers, les rectangles arrondis représentent les outils. Trois outils apparaissent dans le schéma : le 'Static Checker', le 'Proof Obligation Generator' (POG), et le 'Prover'. Ces outils sont lancés en tâche de fond par un scheduleur global appelé 'Project Builder'. Sans rentrer dans le détail, le processus se passe comme suit : le 'Static Checker' reçoit en entrée un ensemble de fichiers qui correspondent à des composants 'unchecked', c'est-à-dire aux modèles et contextes construits par l'utilisateur Il vérifie la consistance de ces entrées – c'est-à-dire qu'elles sont syntaxiquement correctes – et produit un fichier de sortie, qui est un composant 'checked' qui sera passé au 'Proof Obligation Generator'. Le POG génère des obligations de preuve à partir de composants 'checked'. Ces obligations de preuve sont ensuite passées au 'Prover' qui génère des preuves.

Bien que cette présentation puisse paraître simpliste, nous reviendrons sur le processus de preuve dans le chapitre 6.

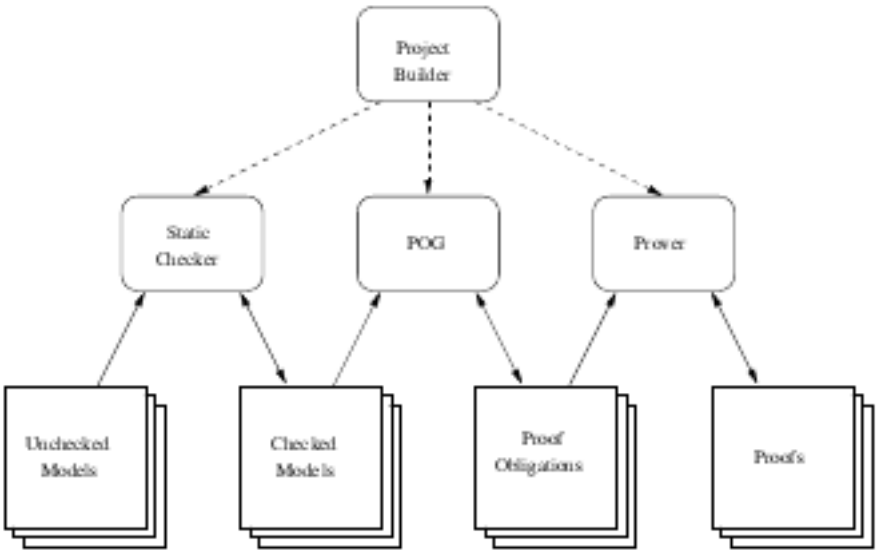

FIG. 5.4 – Architecture de construction

5.3.3 Représentation et status des obligations de preuve

Un exemple de l'ensemble des entrées d'obligations de preuves suit.

Exemple 5.1. Chaque obligation de preuve est indexée par son identificateur unique, ici $m.inv$. Le 'type environment' des variables x et x' est `entier`.

$$m.inv : x \mapsto \mathbb{Z}, x' \mapsto \mathbb{Z} \cdot x \in \mathbb{N}, x' = x + 1 \vdash x' \in \mathbb{N}$$

L'outil RODIN associe automatiquement un *status* à chaque obligation de preuve. Ces status permettent de déterminer si une obligation de preuve a déjà été prouvée entièrement, ou seulement partiellement, ou pas du tout. Les status sont :

1. **Nouveau** : aucune preuve n'a encore été essayée.
2. **En attente** : une preuve a été essayée mais l'obligation de preuve n'a pu être *déchargée*. Il y a au moins un sous-but en attente.
3. **En attente avec lemmes** : l'obligation de preuve a pu être déchargée, mais en utilisant des lemmes non prouvés. Il n'y a plus de but en attente et il existe au moins un lemme dans l'ensemble des sous-buts restants.
4. **Vérifiée** : l'obligation de preuve a été déchargée, mais en utilisant des lemmes vérifiés. Tous les sous-buts restants sont marqués vérifiés.
5. **Complet** : l'obligation de preuve a été déchargée. Il n'y a plus de sous-buts.

5.3.4 Application

Dans cette application, nous montrons comment modéliser un problème – ici la traversée du pont – en utilisant l'outil RODIN. Dans la suite, nous ne détaillerons pas la phase de modélisation : nous reprenons la solution apportée dans la sous-section 4.4.1 et l'intégrons dans RODIN en créant le modèle et le contexte adéquat. La figure 5.2 page 70 illustre la modélisation du modèle `Pont`.

Il est par contre plus intéressant de s'attarder à la phase de vérification. Celle-ci se passe, rappelons-le, dans la perspective 'Proving' de l'outil. La vue 'Obligation Explorer' contient l'ensemble des obligations de preuve que nous devons décharger. Les status de ces dernières sont *Nouveau* (un point d'interrogation dans un rond blanc) ou *En attente* (un point d'interrogation dans un rond rouge).

Pour effectivement tenter de décharger ces obligations, il faut utiliser un plugin externe. Nous utiliserons ici celui de b4free[3], et plus particulièrement le *prover* pour les prédicats sur les hypothèses sélectionnées (dénotées par $p0$ dans l'interface). Utilisons donc ce plugin pour prouver le séquent tel que défini dans l'équation (5.2). L'identifiant de cette obligation est `INITIALISATION/inv2/INV` : `INITIALISATION` est le nom de l'événement en cours, `inv2` correspond à l'invariant qui entre en compte (ici, $temps \in \mathbb{N}$) et `INV` veut dire que nous sommes intéressés par la préservation de l'invariant. L'entrée qui est passée au programme qui a exécuté la preuve (dans le cas de b4free, il s'appelle `krt`) est :

Flag(FileOn("/tmp/eventbou47085.tmp")) & Set(toto | (0 : NATURAL))

Nous retrouvons bien, dans cette entrée le but $0 \in \mathbb{N}$ et pouvons également constater l'absence d'hypothèses, ce qui correspond bien à l'équation (5.2). La preuve étant évidente, le programme retourne *Succès* et l'obligation de preuve est prouvée.

[3]Ce plugin, ainsi que d'autres, sont disponibles à l'adressse suivante : `http://rodin-b-sharp.cvs.sourceforge.net/rodin-b-sharp/`

Troisième partie

Développement

Chapitre 6

Disprover plugin

Dans les chapitres précédents, nous nous sommes attachés à la description de deux langages de modélisation : le B classique et le B événementiel. Nous avons vu comment construire des spécifications et comment générer des obligations utilisées pour la preuve de tels systèmes. Cependant, plus la spécification devient complexe, plus le nombre d'obligations de preuve à décharger augmente. Alors que beaucoup de ces obligations peuvent être déchargées automatiquement par des outils tels que RODIN, un nombre considérable reste à prouver de manière interactive. Cela peut soit être dû à une preuve trop complexe ou parce que le modèle B est erroné. Dans ce chapitre, nous décrivons un *disprover plugin* pour RODIN qui utilise ProB pour trouver automatiquement des contre-exemples pour une obligation de preuve donnée problématique. Dans le cas où le disprover trouve un contre-exemple, l'utilisateur peut directement investiguer la source du problème (comme précisée par le contre-exemple) et ne doit pas essayer de prouver l'obligation. Nous discuterons également sous quelles circonstances notre plugin peut être utilisé comme *prover* c'est-à-dire quand l'absence de contre-exemple est en fait une preuve de l'obligation de preuve.

Ce chapitre est directement tiré d'une traduction libre de [LBL07].

6.1 Introduction

Comme nous l'avons vu dans le chapitre 5, RODIN est accompagné de provers automatiques – tels que le plugin de b4free – qui peuvent décharger un nombre considérable d'obligations de preuve automatiquement. Evidemment, dû à l'incomplétude, toutes les preuves ne peut être automatiques. Dans ces circonstances, l'utilisateur est délaissé, se demandant :

- L'obligation de preuve est-elle valide et la preuve simplement trop compliquée pour le prover automatique ? En d'autres mots, dois-je lancer le prover interactif et essayer de prouver le but manuellement ?
- Où y a-t-il un problème avec la spécification et dois-je investiguer du

temps pour trouver l'erreur et ensuite la corriger ?

Poursuivre l'une ou l'autre voie peut mener à une perte d'efforts considérables. Dans ce chapitre, nous présentons un outil qui aide l'utilisateur dans cette situation précise : un disprover qui essaye de trouver un contre-exemple pour l'obligation de preuve difficile et problématique (voir la figure 6.1).

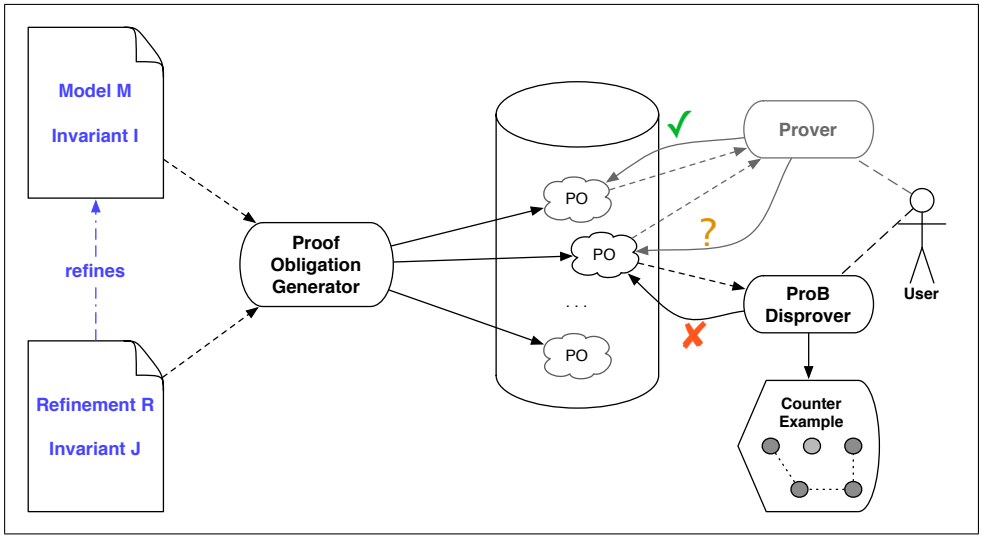

FIG. 6.1 – Vue d'ensemble des obligations de preuve et du rôle du disprover

- Si le disprover trouve un contre-exemple, nous savons qu'il est inutile de passer du temps avec le prover interactif. Le contre-exemple nous donnera également un 'handle' sur le problème et nous aidera à trouver l'erreur dans la spécification plus rapidement.
- Si le disprover ne trouve pas de contre-exemple, nous savons que – sous certaines circonstances du moins – l'obligation de preuve semble être valide. Bien sûr, nous ne sommes toujours pas sûrs que l'obligation de preuve est vraie dans toutes les circonstances ; mais nous avons au moins gagné plus de confiance dans sa validité.

Comme exemple, supposons que nous voulons prouver le théorème selon lequel chaque graphe fini non-dirigé a au moins deux noeuds de degrés égaux.[1] Notre modèle B événementiel contiendra l'ensemble basique $NODES$ et un graphe qui consiste en un ensemble $V \subseteq NODES$ de vertex – plus couramment appelé sommets en français – ainsi que d'une relation binaire symétrique E représentant les arcs. Le degré d'un sommet v est simplement $card(\{v\} \lhd E) = card(E[\{v\}])$. Un prédicat représentant notre théorème

[1] Cet exemple s'inspire d'une conférence donnée par Leslie Lamport au B'2007.

pourrait être de la veine de ce qui suit :

$$V \subseteq NODES \land E \in NODES \leftrightarrow NODES \land E = E^{-1} \land card(V) \in \mathbb{N}$$
$$\Rightarrow$$
$$\exists x \exists y : x \in V \land y \in V \land x \neq y \land card(\{x\} \lhd E) = card(\{y\} \lhd E)$$

Cependant, ce théorème n'est pas prouvable puisque nous avons injecté deux erreurs dans la définition. Alors qu'il n'est pas évident de déceler les erreurs commises, le disprover plugin trouve des contre-exemples qui nous aideront à identifier les problèmes et à corriger le théorème. Un contre-exemple trivial que l'outil trouve est le graphe vide ; un autre contre-exemple trouvé par notre outil avec 5 noeuds qui contiennent des boucles est montré à la figure 6.2. Comme nous pouvons le constater, tous les noeuds ont des degrés différents. Nous devons donc renforcer la partie gauche de notre implication pour refuser les boucles et les graphes avec moins de deux noeuds, après quoi le disprover ne peut plus trouver de contre-exemple :

$$V \subseteq NODES \land E \in NODES \leftrightarrow NODES \land E = E^{-1} \land card(V) \in \mathbb{N} \land$$
$$card(V) > 1 \land id(NODES) \cap E = \varnothing$$
$$\Rightarrow$$
$$\exists x \exists y : x \in V \land y \in V \land x \neq y \land card(\{x\} \lhd E) = card(\{y\} \lhd E)$$

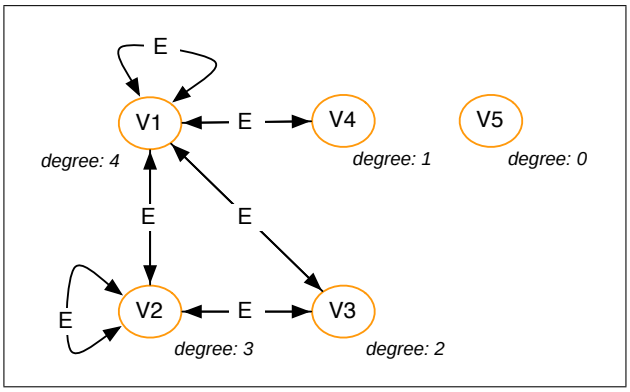

FIG. 6.2 – Contre-exemple trouvé par le disprover

RODIN peut être étendu par des parties tierces, en particulier il est possible d'ajouter des outils de preuve externes. C'est dans cette optique que nous avons développé un prover plugin, qui agit en fait comme un disprover. Notre plugin est basé sur l'animateur et model checker PROB.

L'idée principale de notre travail est d'utiliser l'animateur PROB pour trouver des contre-exemples pour une obligation de preuve individuelle. Par conséquent, nous transformons cette obligation de preuve en une machine

B comme décrit dans la section 6.3. Bien sûr, on aurait pu utiliser le model checker de ProB sur l'entièreté du modèle B événementiel. C'est une option alternative de validation, mais elle va "seulement" trouver les séquences d'opérations qui violent l'invariant en commençant par une initialisation valide, c'est-à-dire qu'elle ne va pas détecter de problèmes si l'invariant est trop faible (voir [LB03]). De plus, en concentrant notre attention sur une seule obligation de preuve problématique, nous pouvons augmenter la probabilité du disprover de trouver des contre-exemples.

Le reste du chapitre est structuré comme suit. D'abord nous fournissons quelques rappels sur la preuve en B événementiel dans la section 6.2. Ensuite, nous présentons la méthodologie sous-jacente de notre disprover plugin dans la section 6.3, avant de discuter l'implémentation effective dans la section 6.4. Nous concluons ce chapitre avec des remarques sur la manière d'utiliser le disprover comme prover dans la section 6.5.

6.2 Le sous-système de preuve de RODIN

Dans RODIN, un nombre considérable de preuves peut être réalisé automatiquement par le sous-système de preuve qui consiste, comme montré dans la figure 6.3, du 'Proof Obligation Generator' et du 'Event-B Kernel Prover' [ROD05a]. Le 'Proof Obligation Generator' extrait toutes les obligations du modèle B événementiel qui doivent être déchargées afin de prouver la validité du modèle et les enregistre dans un fichier de base de données XML. Après que toutes les obligations de preuve ont été générées, le 'Kernel Prover' essaye de décharger des POs valides automatiquement, c'est-à-dire en tâche de fond, sans interactions de l'utilisateur.

Comme montré dans la figure 6.3, le 'Event-B Kernel' se décompose en un 'Proof Manager' et en un ensemble de 'prover plugins'. Alors que le 'Proof Manager' est responsable du stockage, de la traversée, de la composition et de la réutilisation des preuves, les 'prover plugins' essayent de générer des inférences valides dans le but de décharger les obligations de preuve. Le 'Proof Manager' maintient également l'état des preuves actuelles pour toutes les obligations de preuve, décide si elles ont été déchargées et appelle les provers externes si elles sont non-interactives. Il y a également des 'interactive reasoners' qui demandent d'être appliqués par l'utilisateur, le *disprover plugin* ProB est un tel *plugin* interactif. Dans la section qui suit, nous montrons les principes sous-jacents de notre plugin, avant de montrer dans la section 6.4 comment il a été intégré à la plateforme RODIN.

6.3 Le principe du *disproving* utilisant ProB

Dans ce qui suit, nous expliquons comment un séquent peut être transformé en machine B qui peut être utilisé par ProB.

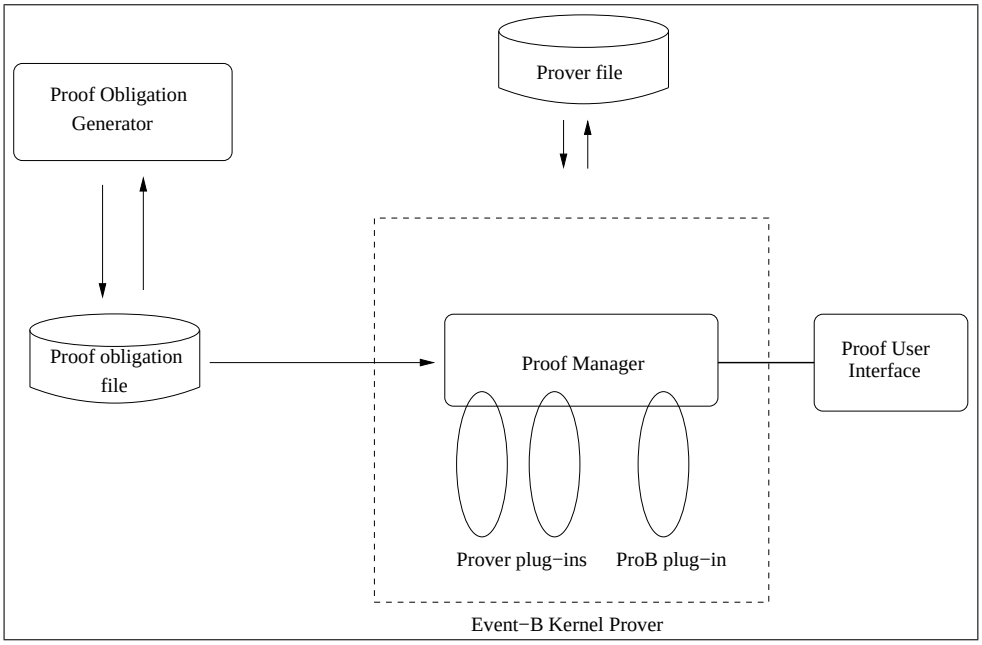

FIG. 6.3 – Architecture du sous-système de preuve de RODIN

6.3.1 Découverte des contre-exemples

Soit $G(x_1, ..., x_k)$ le but d'un séquent s et soit $H_1(x_1, \ldots, x_k), \ldots,$ $H_n(x_1, \ldots, x_k)$ les hypothèses. Pour trouver un contre-exemple de s, nous devons vérifier si le prédicat

$$\exists x_1, ..., x_k : (H_1(x_1, ..., x_k) \wedge ... \wedge H_n(x_1, ..., x_k)) \wedge \neg G(x_1, ..., x_k) \qquad (6.1)$$

tient. Si c'est le cas, alors nous pouvons extraire un contre-exemple concret en trouvant une 'valuation' pour $x_1, ..., x_k$ qui rend l'implication vraie. Trouver des valeurs qui satisfont une formule booléenne propositionnelle est NP complet, et pour les formules de la logique de premier ordre qui peuvent apparaître des séquents, le problème est indécidable. Pour outrepasser cette difficulté, nous devons restreindre les ensembles à des domaines relativement petits et finis. En conséquence, nous savons qu'en principe, il n'est pas possible de garantir qu'un algorithme 'disprover' peut trouver automatiquement un contre-exemple si ce dernier existe. En d'autres mots, l'absence de contre-exemple ne signifie pas en général que l'obligation de preuve est valide. (Il y a, cependant, certains cas où l'absence de contre-exemple décharge une obligation de preuve. Nous discutons de ces cas dans la section 6.5.)

6.3.2 Transformation des séquents en machines B classiques

ProB peut être utilisé pour trouver un contre-exemple pour un séquent donné, mais il a besoin pour cela d'une machine B classique qui encode le séquent comme entrée. Heureusement, cet encodage n'est – du moins en principe – pas difficile à obtenir. Nous créons une machine B, qui contient une opération *disproveHypotheses* avec le prédicat de l'équation (6.1) comme garde. L'opération est active si et seulement si ProB peut trouver un contre-exemple.

Afin de construire cette machine seule, nous devons extraire certaines informations de la spécification B événementiel originale telles que les axiomes[2], 'carrier sets' paramètres, variables (incluant les informations sur le type) et les constantes. De plus, nous nécessitons des informations à propos du séquent à (dé)-charger, tels que les hypothèses et le but. La transformation de ces informations est dans la plupart des cas directe ; nous construisons par exemple la clause `SETS` de la machine en énumérant les définitions des ensembles de la spécifications B événementiel originale. Dans certains cas, la transformation est moins évidente. Nous transformons par exemple les axiomes ainsi que les informations sur le type des constantes dans la clause `PROPERTIES`. Nous générons de nouvelles définitions appelées `TypeEnvironment` et `Hypotheses` à l'intérieur de la clause `DEFINITIONS`. Le `TypeEnvironment` est un sous-ensemble des hypothèses qui ne contient que les prédicats qui s'occupent des informations sur le type. Un schéma de la machine B construite à partir d'un séquent $H_1, H_2, \ldots H_n \vdash G$ se trouve au listing 6.1 page suivante.

6.3.3 Sélection des hypothèses

Le sous-système de preuve de RODIN autorise l'utilisateur à sélectionner un sous-ensemble des hypothèses qui sont dans la base de données, ces hypothèses sont soit directement dérivées de la spécification ou prouvées auparavant. De manière évidente, si un sous-ensemble de H prouve G, alors H prouve également G.

$$H' \subseteq H \wedge H' \vdash G \Rightarrow H \vdash G$$

Un utilisateur peut donc restreindre les hypothèses d'un séquent à un sous-ensemble arbitraire appelé *selected hypotheses*, en supprimant les hypothèses qui ne sont pas pertinentes pour la preuve. Par défaut, un ensemble particulier d'hypothèses qui s'occupent des variables impliquées sont automatiquement sélectionnées par RODIN. L'utilisateur peut également décider de cacher un sous-ensemble particulier d'hypothèses, ce sous-ensemble étant appelé *hidden hypotheses*. Il existe en fait deux alternatives :

[2] Un axiome est traité comme un séquent $true \vdash A$

Listing 6.1 – Schéma d'une machine abstraite construite à partir d'un séquent

```
1   MACHINE Disprove
2   DEFINITIONS
3     TypEnvironment == H1(x1,...,xk) & ... & Hi(x1,...,xk);
4     Hypotheses == TypeEnvironment &
          Hi+1(x1,...,xk) & ... & Hn(x1,...,xk);
5     Goal == G(x1,...,xk)
6   OPERATIONS
7     disprove(x1,...,xk) =
8       PRE Hypotheses & not(Goal)
9       THEN skip
10      END
11    END
```

– lancer le disprover externe avec les *selected hypotheses* ou
– le lancer avec les *all hypotheses* exceptées celles cachées.

Dans tous les cas, l'utilisateur peut choisir quelle alternative appliquer (notre plugin fournit deux boutons) et changer d'avis plus tard.

6.4 Implémentation du *disprover plugin* ProB

Dans un travail précédent [Ben06] a été développé une version de ProB qui s'intègre à Eclipse. Son composant principal est le 'Eclipse ProB plugin' comme montré à la figure 6.4 page suivante. Il permet à des outils tiers d'utiliser ProB pour différentes tâches ; il peut par conséquent être vu comme une couche d'abstraction Java pour la partie Prolog de ProB. Le *disprover* utilise ce noyau central pour trouver des contre-exemples. Il crée donc – lorsqu'il est appliqué à un noeud de l'arbre de preuve – une machine B comme décrit dans la section 6.3 et commence l'animation de la machine. Si l'opération `disprove` est active, nous avons trouvé un contre-exemple.

Le reste de cette section se divise en trois parties : tout d'abord, nous traçons un portrait de l'architecture générale du *disprover* ; ensuite nous voyons comment les contre-exemples trouvés par le *plugin* sont affichés à l'écran ; finalement, nous décortiquons un exemple complet d'utilisation du *plugin*.

6.4.1 Architecture du *disprover*

La figure 6.5 page suivante donne une vue globale de l'architecture du *disprover plugin*. Comme le montre la figure, le *plugin* se compose d'une

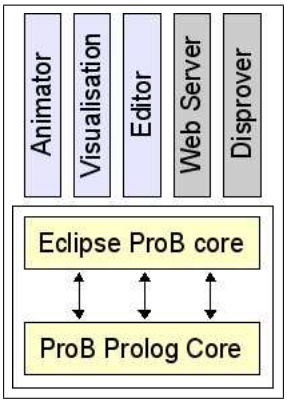

FIG. 6.4 – Architecture de la version Eclipse de PROB

interface graphique[3], le composant *disprover ui plug-in*, qui affiche le résultat d'une preuve et d'un composant central, *disprover core plug-in*, qui encapsule la logique de preuve. L'interface graphique est une extension de l'interface à l'interface graphique de preuve de RODIN. Elle permet à l'utilisateur de sélectionner un noeud de l'arbre de preuve que ce dernier veut vérifier avec PROB. Le plugin central fournit un moyen d'appliquer le disprover PROB. Son rôle est de :

- transformer le séquent en machine B ;
- appeler PROB à travers le 'Eclipse PROB core plugin' ;
- retourner le résultat à l'utilisateur ;
- gérer les erreurs, 'time out' et demande d'annulation de l'utilisateur.

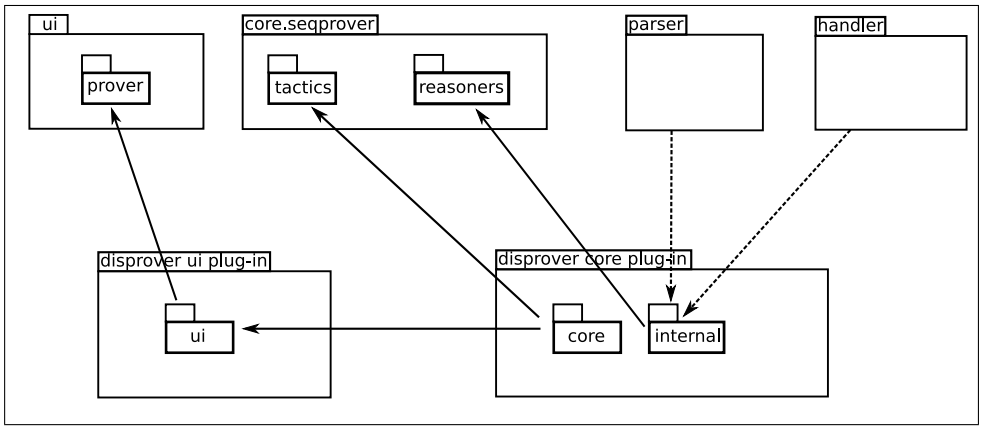

FIG. 6.5 – Architecture générale du *disprover*

[3]couramment appelé User Interface ou UI en anglais

Plus précisément, le composant principal est composé de deux sous-composants : *core* et *internal*. Le composant *core* est le point d'entrée pour le composant *disprover ui plug-in*. Le composant *internal* est, comme son nom l'indique, caché au monde extérieur dans le sens où il n'est pas possible d'y accéder directement. Ce dernier transforme dans un premier temps le séquent reçu en une machine abstraite telle que décrite dans la sous-section 6.3.2. Initialement, la machine est représentée textuellement comme dans le listing 6.1. Cependant, ProB travaille avec des représentations Prolog [LB03] et il faut donc transformer la machine abstraite. Nous utilisons pour cela le composant externe *parser* dont le rôle est de générer une représentation Prolog de la machine abstraite. Nous pouvons à présent interagir avec ProB pour lui demander d'animer la machine. Cette animation s'opère à travers le composant *handler*, qui permet notamment d'initialiser la machine, d'exécuter des opérations et de récupérer la valeur de certaines variables dans l'état courant. Si l'animateur trouve un contre-exemple, les informations concernant celui-ci sont stockées en internes et utilisées pour construire la réponse comme il est expliqué à la sous-section 6.4.2. Finalement, le *plugin* retourne une règle de preuve en utilisant le composant *reasoners*. Cette règle de preuve est en fait une liste de nouveaux séquents à insérer comme fils de l'arbre de preuve.[4]

6.4.2 Affichage des contre-exemples

Une première approche était d'afficher les contre-exemples dans une fenêtre séparée. Cette solution n'était cependant pas très utile parce qu'il n'y a dans ce cas aucune connexion entre le contre-exemple et les obligations de preuve. Nous avons donc choisi une approche alternative pour résoudre ce problème en appliquant une distinction de cas [Abr96] au noeud de l'arbre de preuve.[5] Comme mentionné dans la section précédente, un contre-exemple peut être décrit par un prédicat

$$C_p \equiv x_1 = e_1, ..., x_k = e_k$$

Nous appliquons à présent la distinction de cas sur le noeud. Cela résulte en deux noeuds fils avec les séquents

1. $H, C_p \vdash G$
2. $H, \neg C_p \vdash G$

Le premier séquent est le cas dans lequel le contre-exemple a été trouvé (C_p rend G faux). Le second est le cas restant, où le contre-exemple spécifique est explicitement rejeté. Ce second cas peut être utilisé pour trouver des contre-exemples supplémentaires en réappliquant le disprover sur ce dernier.

[4] Voir à ce sujet la sous-sectionsous-section 4.1.2 page 46.
[5] Idée originale de Farhad Mehta

6.4.3 Exemple d'utilisation du plugin

Il est à présent temps de voir une utilisation concrète du plugin sur un exemple. La simplicité de l'exemple permet de se focaliser sur l'agencement des différentes étapes nécessaires au bon déroulement de la découverte du contre-exemple. L'exemple est tiré de la `IV`^{e} partie de [ROD05b] et s'énonce comme suit : construire un système qui contrôle le passage de voitures sur un pont étroit entre un continent et une île. Deux feux rouges sont situés de chaque côté du pont. On suppose que les voitures ne passent que quand le feu est vert, et attendent leur tour quand le feu est rouge. Il y a également quatre capteurs – deux de chaque côté – qui détectent les voitures qui veulent entrer et sortir de l'île et du continent. Seul un nombre limité de voitures peut se trouver en même temps sur le pont.

La spécification que nous définissons ici est d'un haut niveau. L'île et le pont sont considérés comme une seule entité, et les deux événements consistent à passer du continent à ce couple pont-île et vis-versa. Nous nous concentrons ici sur l'événement `ML_out`, qui permet à une voiture de passer du contient au couple pont-île. La spécification de cet événement est la suivante :

```
1  ML_out
2  WHEN
3     n < d
4  THEN
5     n := n + 1
6  END
```

où n est une variable contenant le nombre de voitures sur le couple pont-île et d est une constante qui dénote le nombre maximum de voitures se trouvant sur le couple pont-île. Une voiture ne peut donc traverser que si le nombre maximum n'est pas encore atteint. Lorsqu'elle traverse, on incrémente le nombre de voitures qui se trouvent sur le couple pont-île d'une unité. Pour prouver une partie de cet événement, nous aurons également besoin de l'invariant de la variable n, qui est $n \in \mathbb{N}$.

La figure 6.6 page 89 représente l'état de RODIN après la modélisation de la solution. Comme on peut le constater sur le screenshot, il reste une et une seule obligation à prouver pour le modèle `m0_island_bridge`. Cette obligation porte le nom `ML_out/inv1/INV` où `ML_out` est le nom de l'événement en cours, `inv2` correspond à l'invariant qui entre en compte (ici, $n \in \mathbb{N}$) et `INV` veut dire que l'obligation porte sur la préservation de l'invariant.

Il est à présent temps de lancer le *disprover*. Juste avant cela, nous devons choisir si nous prenons toutes les hypothèses, ou seulement un sous-ensemble de ces dernières. Considérons le deuxième choix, à savoir les *selected hypotheses*. La seule hypothèse que nous pouvons sélectionner est $n < d$. Le

séquent construit par RODIN et associé à l'obligation est :

$$n < d \vdash n + 1 \in \mathbb{N}$$

Ce séquent est passé au *disprover plugin*, qui le transforme en la machine abstraite suivante :

Listing 6.2 – Machine abstraite construite à partir du séquent

```
1  MACHINE BMachine
2
3  DEFINITIONS
4    SET_PREF_MININT == -5;
5    SET_PREF_MAXINT == 5;
6    SET_PREF_MAX_OPERATIONS == 1;
7    SET_PREF_MAX_INITIALISATIONS == 1;
8    TypEnvironment == n ∈ ℤ;
9    Hypotheses == TypEnvironment ∧ (n < d);
10   Goal == ((n + 1) ∈ ℕ)
11
12 CONSTANTS
13   d
14
15 PROPERTIES
16   (d ∈ ℕ) ∧ (d > 0)
17
18 VARIABLES varn
19
20 INVARIANT varn ∈ ℤ
21
22 INITIALISATION
23   ANY n WHERE Hypotheses
24   THEN varn := n
25   END
26
27 OPERATIONS
28   disproveHypotheses(n) =
29     PRE Hypotheses ∧ not(Goal)
30     THEN varn := n
31     END
32 END
```

Les quatre premières définitions sont des préférences pour ProB. Les deux premières déterminent les valeurs minimales et maximales des entiers implémentables, c'est-à-dire ceux définis par `INT` au lieu de `INTEGER`. Ces valeurs peuvent être changées dans les préférences de RODIN. Les deux

suivantes limitent le choix de l'initialisation et des opérations à un. Cette limitation permet de s'affranchir de la sélection de l'initialisation ou d'une opération lorsqu'elles utilisent le non-déterminisme, comme dans l'initialisation de la machine abstraite ci-dessus. Le but du séquent apparaît dans la clause `DEFINITION` de la machine, sous le nom *Goal*. Le type de la variable n est renseigné dans le *TypEnvironment*. Les hypothèses prennent en compte l'hypothèse sélectionnée $n < d$, où le type de la constante d est donné dans les propriétés. Nous définissons ensuite une nouvelle variable *varn* qui contiendra la valeur du contre-exemple une fois celui-ci trouvé.[6] *varn* est initialisée avec une valeur qui respecte les *Hypotheses*. L'opération *disproveHypotheses* permet d'enregistrer la valeur du contre-exemple dans le cas où les hypothèses sont vérifées et le but est faux.

ProB reçoit ensuite comme entrée la machine abstraite et peut commencer l'animation. La première étape est l'établissement des constantes : la constante d reçoit la valeur 1, ce qui se note par l'opération `setup_constants(1)`. La deuxième étape est l'initialisation de la machine : la variable *varn* est initalisée à -5, ce que dénote l'opération `initialise_machine(-5)`. La troisième étape est l'exécution de l'opération `disproveHypotheses(-5)`, qui assigne la valeur -5 à la variable *varn*.[7] L'animation est maintenant terminée et le contre-exemple est disponible dans la variable *varn*.

Finalement, la dernière tâche du *plugin* est de récupérer le contre-exemple et de l'afficher à l'utilisateur. Deux séquents sont construits à partir du contre-exemple sur base du principe de la sous-section 6.4.2 :

1. $n < d \land n = -5 \vdash n + 1 \in \mathbb{N}$
2. $n < d \land \neg(n = -5) \vdash n + 1 \in \mathbb{N}$

Le premier séquent contient le contre-exemple trouvé : il est évidemment faux. Le deuxième séquent dénote le cas où le contre-exemple a été rejeté ($\neg(n = -5)$). Il est dès lors possible de ré-appliquer le *disprover* sur ce séquent pour vérifier si un autre contre-exemple ne peut pas être trouvé.

L'état de RODIN après l'application du *disprover* se trouve à la figure 6.7 page 90. La vue *Proof Tree* en haut à gauche contient les informations relatives à l'application du *plugin*, à savoir le contre exemple trouvé ($n = -5$) et la distinction de cas. Les différentes hypothèses et le but des deux séquents sont disponibles dans la fenêtre centrale de RODIN.

[6] Cette définition est nécessaire car l'API *java* de ProB ne permet pas d'accéder directement au contre-exemple.

[7] Notons que la valeur de la variable *varn* était déjà de -5.

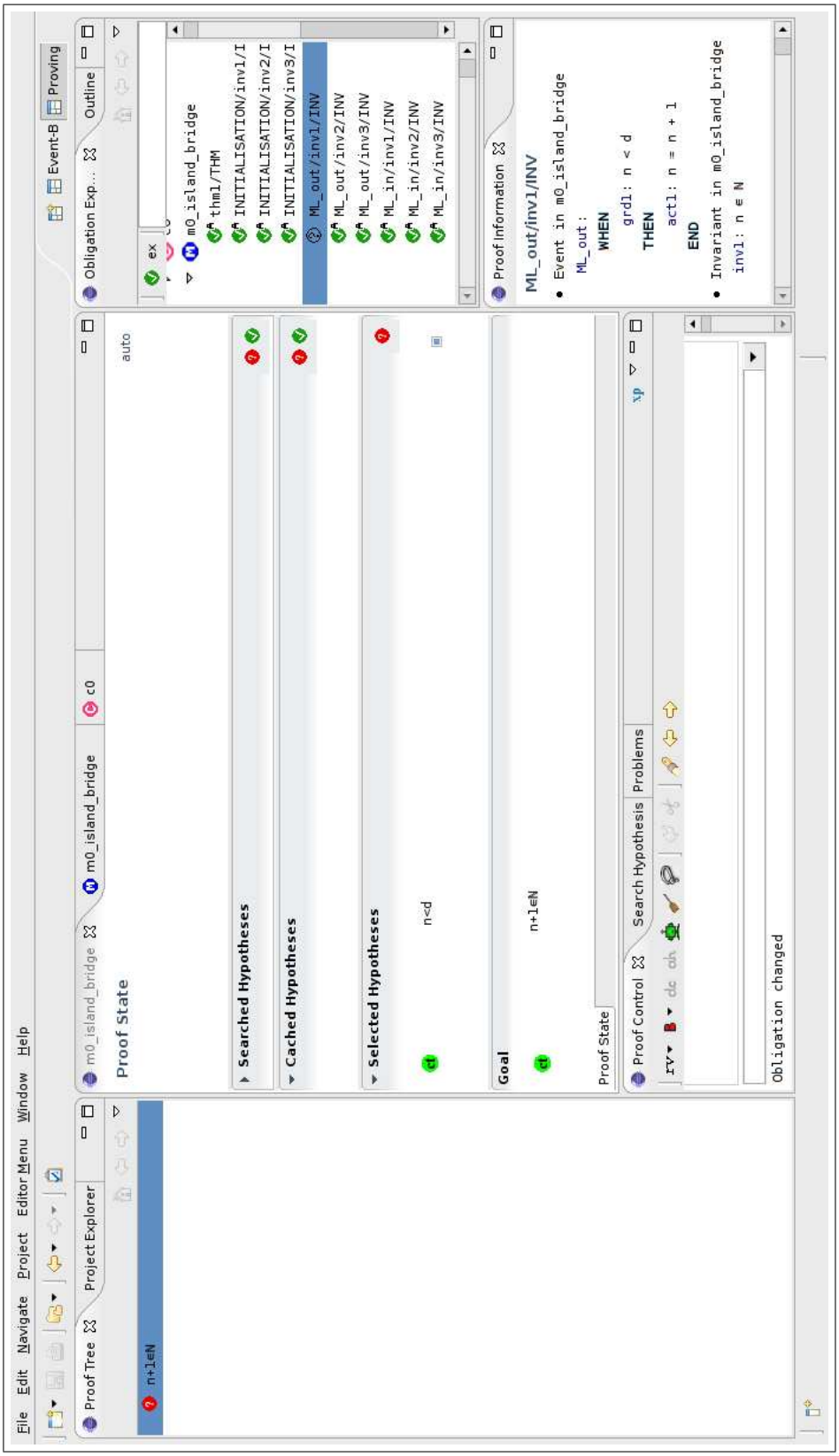

FIG. 6.6 – Screenshot de l'état de RODIN après la modélisation

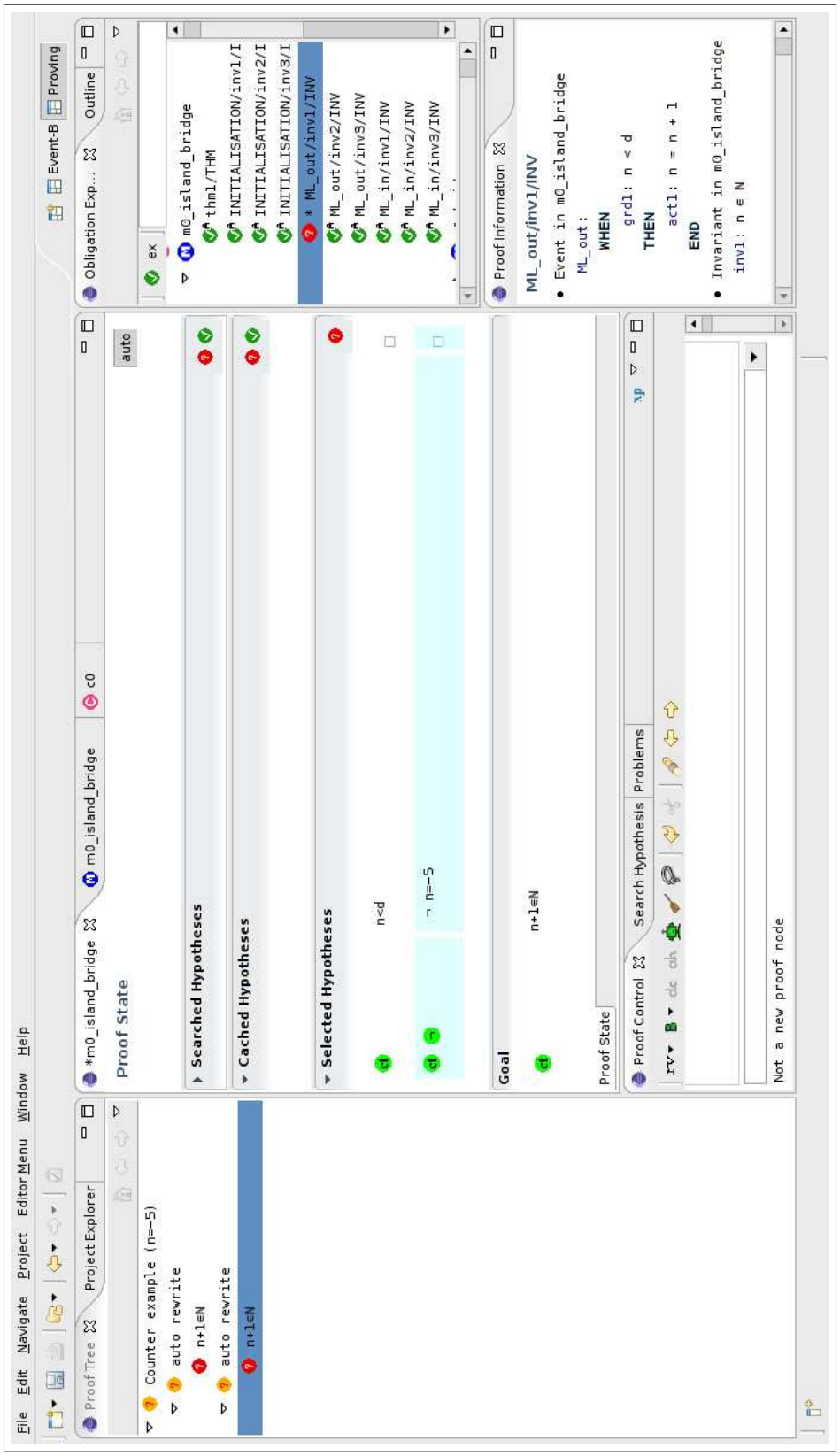

FIG. 6.7 – Screenshot de l'état de RODIN après l'application du *disprover*

6.5 Perspectives futures

6.5.1 Utilisation de ProB comme *prover*

Si le *disprover* ProB ne réussit pas à trouver un contre-exemple pour une obligation de preuve particulière, nous ne pouvons pas inférer que l'obligation de preuve est vraie. Cela est du à deux raisons :

- ***deffered sets :*** Si une machine B utilise des *deffered sets* (c'est-à-dire des ensembles qui ne sont pas explicitement énumérés dans la clause `SET`), alors la cardinalité de ces ensembles n'est pas fixée à priori ; l'ensemble pourrait même être infini. Cependant, ProB va vérifier les obligations de preuve seulement dans le cas où la cardinalité des *deffered sets* est finie, et pourrait donc échouer à trouver des contre-exemples existants. Par exemple, ProB échouera à trouver un contre-exemple pour la formule $\exists n.(n : \mathbb{N} \wedge card(S) < n)$, où S est un *deffered set* sans autres restrictions.
- **entiers :** Si une valeur entière apparaît dans une obligation de preuve, et dont la valeur n'est *pas* déterminée par le reste de l'obligation de preuve, ProB énumérera la variable seulement dans un intervalle fini (entre les valeurs MININT et MAXINT déterminées par l'utilisateur). A nouveau, ProB peut échouer à trouver des contre-exemples pour des valeurs entières qui se situent en dehors de MININT..MAXINT.

Pourtant, si une machine B ne contient ni des *deffered sets* ni des variables entières, le ProB *disprover* peut en fait aussi être utilisé comme *prover*. Cette condition peut facilement être vérifiée de manière statique, dans quel cas notre *disprover* peut informer la plateforme RODIN que la PO a en réalité été prouvée. Quelques spécifications B pratiques tombent dans cette catégorie. Citons par exemple la fonction de voiture Volvo utilisée dans [LB03]. Un autre exemple est l'encodage de Hamming [CHO04], pour lequel Dominique Cansell a utilisé ProB pour prouver quelques théorèmes essentiels (qui auraient été extrêmement fastidieux à prouver manuellement).[8]

6.5.2 Autres perspectives

Une évaluation empirique de l'utilisation de ProB comme *disprover* est exigée si l'on désire mesurer l'efficience du *plugin*, ou si ce dernier peut être optimisé. Un grand nombre de tests ont déjà été faits mais des tests de benchmark sur différents systèmes d'exploitation seraient les bienvenus.

Lors de l'utilisation de relations ou de fonctions dans le B événementiel, les variables d'un séquent augmentent très largement. Par exemple, étant donné $r : A \leftrightarrow A$, où la cardinalité de A est 4, on a $2^{4*4} = 65536$ possibilités pour r. Etant donné $x : \mathbb{P}(A \leftrightarrow A)$, nous avons $2^{2^{4*4}} = 2^{65536}$ possibilités pour x. ProB doit investiguer ces possibilités pour la recherche d'un contre-

[8] Communication privée de Dominique Cansell.

exemple. La réduction symétrique est une façon de faciliter cette tâche, et il serait approprié de vérifier si nous pouvons utiliser les développements récents de ProB [LM07] dans cette optique pour notre *disprover plugin*. Une option alternative est de partitionner l'espace de configuration en plusieurs zones, et laisser différentes instances de ProB tourner en parallèle pour prendre soin de l'exploration correspondante.

6.5.3 Travaux apparentés

Un outil très populaire pour la validation de modèles et la découverte de contre-exemple est Alloy [Jac02], qui utilise des solvers SAT (à la place de 'constraint solving'). Le langage de spécification Alloy est de premier ordre et ne peut donc pas être appliqué 'out of the box' aux modèles B événementiels. Cependant, les modèles B événementiels peuvent être compilés (pour des ensembles finis) en Alloy ou directement en formules SAT.

Comme travaux apparentés précédents, on trouve les générateurs de modèle FINDER [SLM94] et MACE [McC01] qui peuvent également être utilisés pour trouver des contre-exemples. Le *prover* Isabelle a maintenant également une fonction rapide de 'ckeck' [BN04], recherchant aléatoirement des contre-exemples. Il y a encore beaucoup d'autres travaux connexes, tels que le récent [SBD02], et il y a même certains 'workshops' tels que CADE et IJCAR qui ont été organisés autour du *disproving*. Des travaux considérables se penchent sur la combinaison du *model checking* [CGP99] avec des théorèmes de preuve en général (par exemple, [GP05]).

Chapitre 7

Conclusion et perspectives

Dans ce mémoire, nous avons présenté un *disprover plugin* pour RODIN qui contribue à la phase de preuve dans le développement d'une spécification. Ce *plugin* essaye de trouver des contre-exemples pour chaque obligation de preuve, et dans l'affirmative renvoit le contre-exemple en question pour que l'utilisateur puisse investiguer la source du problème. L'utilisation d'un tel *plugin* peut se révéler utile lorsque l'utilisateur est "bloqué" en attente d'une preuve qui ne peut pas être réalisée par un *prover* externe. Nous avons également vu dans quelle mesure notre *disprover* peut être utilisé comme *prover*, c'est-à-dire quand l'absence de contre-exemples est en fait une preuve. Le but de notre *plugin* n'est nullement de concurrencer ou de remplacer les *provers* externes existants, mais plutôt d'être un outil complémentaire, permettant notamment d'éliminer le temps nécessaire requis à tenter de prouver une obligation de preuve fausse.

Notre *plugin* étant toujours en phase de développement, celui-ci souffre de limitations.

- Lors de l'utilisation de relations ou de fonctions dans des séquents, nous constatons une explosion combinatoire de l'espace d'états lors de la recherche de contre-exemples. Deux solutions sont proposées pour la suite du développement : la première consiste à utiliser la réduction symétrique telle que présentée dans [LM07]. La deuxième suggère de partitionner l'espace d'états en plusieurs zones et d'avoir plusieurs instances de ProB qui tournent en parallèle, chaque instance s'occupant d'une zone particulière.
- Aucune mesure d'efficacité du *plugin* en terme de temps n'a été réalisée. Bien qu'un nombre de tests aient été effectués pour valider les découvertes effectives de contre-exemples, il serait approprié de procéder à des tests de *benchmarks* – notamment sur différents systèmes d'exploitation – pour détecter d'éventuels *bottlenecks* dans l'implémentation du plugin.

Venons-en à présent à la question de la transformation d'un langage

dans un autre. Notre approche a été de partir d'un séquent écrit en B événementiel, et de traduire ce dernier en une machine abstraite qui respecte le B classique pour pouvoir ensuite être utilisée dans un outil tel que ProB. Nous n'avons pas rencontré de problèmes majeurs lors de la phase de transformation mais nous n'avons pas pu parcourir l'entierté du langage B événementiel. D'autre part, notre utilisation s'est limitée à la création d'une machine abstraite dont le but est de découvrir des contre-exemples. Nous n'avons dès lors pas testé l'équivalence d'une spécification complète écrite dans les deux langages, bien que l'exemple de la traversée du pont ait pu être décrite dans les deux approches formelles. De plus, notre connaissance des deux langages n'était pas assez poussée pour investir une telle tâche. Un travail futur dans ce domaine serait l'étude de l'équivalence entre ces deux langages, et par extension la traduction d'un langage dans un autre et les limites inhérentes associées. Ce travail permettrait de faciliter la conversion d'un langage dans un autre et d'augmenter l'interopérabilité par la mise en évidence des concepts similaires.

Annexe A

Définitions mathématiques

Nous rappelons ici quelques définitions mathématiques élémentaires. Ces définitions sont directement tirées de [Rob05]. Nous donnons également les notations permettant de représenter celles-ci en B classique.

A.1 Définitions mathématiques

A.1.1 Relation

Définition A.1 Relation

Une *relation* $S \leftrightarrow T$ est définie par :

$$S \leftrightarrow T = \mathbb{P}(S \times T)$$

Une relation $S \leftrightarrow T$ entre un ensemble S et un ensemble T lie des éléments de S aux éléments de T. Elle n'est rien d'autre que l'ensemble de toutes les paires (s, t) qui sont liées, où $s \in S$ et $t \in T$. Une paire se note $(s \mapsto t)$.

Domaine et rang

Définition A.2 Domaine

Le *domaine* $\mathrm{dom}(r)$ d'une relation r se définit par :

$$\mathrm{dom}(r) = \{x \mid \exists y \cdot x \mapsto y \in r)\}$$

$\mathrm{dom}(r)$ est l'ensemble des éléments de S dont r les lie à quelque chose dans T.

Définition A.3 Rang

Le *rang* $\text{ran}(r)$ d'une relation r se définit par :

$$\text{ran}(r) = \{y \mid \exists x \cdot x \mapsto y \in r\}$$

$\text{ran}(r)$ est l'ensemble des éléments de T qui sont liés à quelque chose dans S.

A.1.2 Image,inverse et composition

Définition A.4 Image relationnelle

Soit r une relation et S un ensemble. Alors l'*image relationnelle* $r[S]$ est définie par :

$$r[S] = \{y \mid \exists x \cdot x \in S \land x \mapsto y \in r\}$$

Notons également que $r[S] = \text{ran}(S \lhd r)$.

Définition A.5 Inverse relationnel

Soit une relation r. Alors l'*inverse relationnel*, dénotée par r^{-1} est définie par :

$$r^{-1} = \{y, x \mid x \mapsto y \in r\}$$

Définition A.6 Composition relationnelle

La *composition relationnelle* de deux relations $r \in S \leftrightarrow T$ et $q \in T \leftrightarrow U$, notée $p\,;q$ est définie par :

$$p\,;q = \{x, y \mid (\exists z \cdot x \mapsto z \in p \land z \mapsto y \in q)\}$$

A.1.3 Fonction

Les fonctions sont des cas particuliers de relations : toutes les propriétés que nous avons vu pour les relations sont donc d'application pour les fonctions.

Définition A.7 Fonction

Soient deux ensembles A et B. Une *fonction de A dans B* est un sous-ensemble $f \subseteq A \times B$ tel que si $(a, b) \in f$ et $(a, b') \in f$ alors $b = b'$. L'ensemble de toutes les fonctions de A dans B est noté $A \rightarrow B$.

Fonctions particulières

Définition A.8 Fonction partielle

Soient S et T deux ensembles et $id(T) = \{x, y \mid x \in T \land y \in T \land x = y\}$. Une *fonction partielle* $S \nrightarrow T$ est telle que :

$$S \nrightarrow T = \{r \mid r \in S \leftrightarrow T \land r^{-1} ; r \subseteq id(T)\}$$

Une fonction partielle $f : S \nrightarrow T$ lie un élément de S à **au plus un** élément de T. Dans le cas où f est une variable, $S \nrightarrow T$ donne également son type.

Définition A.9 Fonction totale

Une *fonction totale* $S \rightarrow T$ est telle que :

$$S \rightarrow T = \{f \mid f \in S \nrightarrow T \land dom(f) = S\}$$

Une fonction totale $f : S \rightarrow T$ lie un élément de S à **exactement un** élément de T

Définition A.10 Fonction injective (partielle ou totale)

Une *fonction injective partielle* $S \rightarrowtail T$ est telle :

$$S \rightarrowtail T = \{f \in S \nrightarrow T \land f^{-1} \in T \nrightarrow S\}$$

Une *fonction injective totale* $S \rightarrowtail T$ est telle :

$$S \rightarrowtail T = S \rightarrowtail T \cap S \rightarrow T$$

Définition A.11 Fonction surjective (partielle ou totale)

Une *fonction surjective partielle* $S \twoheadrightarrow T$ est telle :

$$S \twoheadrightarrow T = \{f \in S \nrightarrow T \land \mathrm{ran}(f) = T\}$$

Une *fonction surjective totale* $S \twoheadrightarrow T$ est telle :

$$S \twoheadrightarrow T = S \twoheadrightarrow T \cap S \rightarrow T$$

Définition A.12 Fonction bijective

Une *fonction bijective* $S \rightarrowtail\!\!\!\!\!\twoheadrightarrow T$ est telle que :

$$S \rightarrowtail\!\!\!\!\!\twoheadrightarrow T = S \rightarrowtail T \cap S \twoheadrightarrow T$$

Une fonction bijective est une fonction totale, injective et surjective. C'est une fonction *one-one* : chaque élément de S est lié à un et un seul élément de T.

A.2 Notations du B classique

Le tableau A.1 donne l'équivalent des notations mathématiques en B classique.

Notations mathématiques	**Notations en B classique**
$S \leftrightarrow T$	`S <-> T`
$\mathrm{dom}(r)$	`dom(r)`
$\mathrm{ran}(r)$	`ran(r)`
$r[S]$	`r[S]`
r^{-1}	`r~`
$p\,;q$	`p ; q`
$S \nrightarrow T$	`S +-> T`
$S \rightarrow T$	`S --> T`
$S \rightarrowtail T$	`S >+> T`
$S \rightarrowtail T$	`S >-> T`
$S \twoheadrightarrow T$	`S +->> T`
$S \twoheadrightarrow T$	`S -->> T`
$S \rightarrowtail\!\!\!\!\!\twoheadrightarrow T$	`S >->> T`

Tab. A.1 – Equivalent des notations mathématiques et en B classique

Annexe B

Précondition la plus faible

Cette annexe explicite les règles de la précondition la plus faible énoncée à la sous-section 3.1.1.

Instruction vide et construction simple

Le corps d'une opération peut être composé, dans sa forme la plus simple, d'une instruction vide (*skip*) ou d'une construction simple de la forme **BEGIN** S **END** où S est une assignation. Les règles suivantes sont d'application :

Propriété B.1 Précondition pour une instruction vide ou une construction simple

Soit P un prédicat et S une assignation. On a alors

$$[skip]P = P$$
$$[\textbf{BEGIN } S \textbf{ END}]P = [S]P$$

Assignations simples et multiples

Il existe deux types d'assignations en B classique : l'assignation simple et l'assignation multiple. Attachons-nous d'abord aux assignations simples.

Propriété B.2 Précondition pour une assignation simple

Soit P un prédicat sur les valeurs finales et $P[E/x]$ un prédicat sur les valeurs initiales où les occurrences libres de x ont été remplacées par l'expression E. Soit $x := E$ l'assignation de l'expression E à la variable x. La règle pour l'*assignation simple* est la suivante :

$$[x := E]P = P[E/x]$$

Passons ensuite aux assignations multiples.

Propriété B.3 Précondition pour une assignation multiple

Soit P un prédicat sur les valeurs finales et $P[E, F/x, y]$ un prédicat sur les valeurs initiales où les occurrences libres de x et y ont respectivement été remplacées par les expressions E et F. Soit $x, y := E, F$ l'assignation multiple des expressions E et F aux variables x et y respectivement. La règle pour l'*assignation multiple* est la suivante :

$$[x, y := E, F]P = P[E, F/x, y]$$

Choix

La syntaxe du choix en B classique est **CHOICE** S **OR** R **END** où S et R sont des assignations. Il existe également une syntaxe alternative qui est équivalente à la première : $S[]R$. C'est cette dernière que nous utilisons pour énoncer la règle.

Propriété B.4 Précondition pour le choix

Soit P un prédicat. Alors on a

$$[S[]R]P = [S]P \wedge [R]P$$

Assignation avec précondition

Une assignation S peut être sujette à une précondition Q, ce que l'on note **PRE** Q **THEN** S **END** où Q est un prédicat. Il existe également une syntaxe équivalente que nous utilisons par la règle : $Q \mid S$.

Propriété B.5 Précondition pour l'assignation avec précondition

Soit Q et P deux prédicats et S une assignation. Alors on a

$$[Q \mid S]P = Q \wedge [S]P$$

Conditionnelle

Dans ce qui suit, nous considérons que P et Q sont des prédicats et S et R sont des assignations. La syntaxe d'une conditionnelle prend la forme **IF** Q **THEN** S **ELSE** R. Introduisons tout d'abord une nouvelle construction, $Q \Longrightarrow S$, qui dit qu'on peut appliquer l'assignation S seulement si l'état

actuel satisfait à la garde Q. La règle pour la précondition la plus faible de cette construction est :

$$[Q \Longrightarrow S]P = Q \Rightarrow [S]P$$

Remarquons également que **IF** Q **THEN** S **ELSE** $R = (Q \Longrightarrow S)[](\neg Q \Longrightarrow R)$. Dès lors, on peut en déduire que :

$$\begin{aligned}&[(Q \Longrightarrow S)[](\neg Q \Longrightarrow R)]P = [Q \Longrightarrow S]P \wedge [\neg Q \Longrightarrow R]P \\ &= (Q \Rightarrow [S]P) \wedge (\neg Q \Rightarrow [R]P)\end{aligned}$$

La règle pour la construction conditionnelle est dès lors :

Propriété B.6 Précondition pour la construction conditionnelle

Soit P et Q deux prédicats et S et R deux assignations. Alors on a

$$[(Q \Longrightarrow S)[](\neg Q \Longrightarrow R)]P = (Q \Rightarrow [S]P) \wedge (\neg Q \Rightarrow [R]P)$$

Annexe C

ProB : traversée du pont

Le listing C.1 est le contenu du fichier *pont.mch* : celui-ci représente notre solution du problème de la traversée du pont dans le formalisme de ProB.

Listing C.1 – *pont.mch*

```
1  MACHINE Pont
2
3  DEFINITIONS GOAL == riveg={} & rived=PERSONNES & temps
       <=16
4
5  SETS
6    LAMPE = {gauche, droite};
7    PERSONNES = {un, deux, trois, quatre}
8
9  CONSTANTS personnes, TEMPS
10
11 PROPERTIES
12   TEMPS <: NATURAL & TEMPS = {1,2,5,8} &
13   personnes : PERSONNES >-->> TEMPS &
14   personnes = {un|->1, deux|->2, trois|->5, quatre
         |->8}
15
16 VARIABLES lampe, temps, riveg, rived
17
18 INVARIANT
19   lampe : LAMPE & temps : NATURAL &
20   riveg <: PERSONNES & rived <: PERSONNES
21
22 INITIALISATION
23   lampe, temps, riveg, rived := gauche, 0, PERSONNES, {}
24
```

```
25 OPERATIONS
26
27   gauchedroite(p) = PRE p <: riveg & p/={} & card(p)
        <=2 & lampe=gauche
28                     THEN IF personnes[p]/={}
29                          THEN lampe,temps,riveg,rived
                                :=
30                              droite,temps+ max(
                                    personnes[p]) ,riveg-p,
                                    rived\/p
31                          END
32                     END;
33
34   droitegauche(p) = PRE p <: rived & p/={} & card(p)
        <=2 & lampe=droite
35                     THEN IF personnes[p]/={}
36                          THEN lampe,temps,riveg,rived
                                :=
37                              gauche,temps+ max(
                                    personnes[p]) ,riveg\/p
                                    ,rived-p
38                          END
39                     END
40
41 END
```

Annexe D

RODIN : traversée du pont

Les listings D.1 et D.2 donnent la solution de la traversée du pont en B événementiel.

Listing D.1 – Contexte

```
1
2  CONTEXT PontContext
3
4  SETS
5    LAMPE
6    PERSONNES
7
8  CONSTANTS
9    gauche
10   droite
11   un
12   deux
13   trois
14   quatre
15   personnes
16   TEMPS
17
18 AXIOMS
19  lampe = {gauche, droite}
20  gauche ≠ droite
21  PERSONNES = {un, deux, trois, quatre}
22  un ≠ deux
23  un ≠ trois
24  un ≠ quatre
25  deux ≠ trois
26  deux ≠ quatre
27  trois ≠ quatre
28  TEMPS ⊆ ℕ
29  TEMPS = {1,2,5,8}
30  personnes ∈ PERSONNES ⤖ TEMPS
```

```
31  personnes = {un ↦ 1, deux ↦ 2, trois ↦ 5, quatre ↦ 8}
32
33 END
```

Listing D.2 – Modèle

```
1
2  MACHINE Pont
3
4  SEES PontContext
5
6  VARIABLES
7    lampe
8    temps
9    riveg
10   rived
11
12 INVARIANT
13   lampe ∈ LAMPE
14   temps ∈ ℕ
15   riveg ⊆ PERSONNES
16   rived ⊆ PERSONNES
17
18 EVENTS
19
20   INITIALISATION
21     BEGIN
22       lampe := gauche
23       temps := 0
24       riveg := PERSONNES
25       rived := ∅
26     END
27
28   gauchedroite
29     ANY
30       p
31     WHERE
32       p ⊆ riveg
33       p ≠ ∅
34       card(p) ≤ 2
35       lampe=gauche
36       personnes[p] ≠ ∅
37     THEN
38       lampe := droite
39       temps := temps+ max(personnes[p])
40       riveg := riveg \ p
41       rived := rived ∪ p
42     END
43
44   droitegauche
```

```
45      ANY
46         p
47      WHERE
48         p ⊆ riveg
49         p ≠ ∅
50         card(p) ≤ 2
51         lampe=droite
52         personnes[p] ≠ ∅
53      THEN
54         lampe := gauche
55         temps := temps+ max(personnes[p])
56         riveg := riveg ∪ p
57         rived := rived \ p
58      END
59
60 END
```

Bibliographie

[Abr96] Jean-Raymond Abrial. *The B book : assigning programs to meanings*. Cambridge University Press, 1996.

[BCUL99] UK B-Core (UK) Limited, Oxon. *B-Toolkit, On-line manual*, 1999. Available at `http://www.b-core.com/ONLINEDOC/Contents.html`.

[BDMB98] P Behm, P. Desforges, J.-M. Meynadier, and Didier Bert. Météor : An industrial success in formal development. *Lecture notes in computer science*, 1393 :311, April 1998.

[Ben06] Jens Bendisposto. Integration of the ProB modelchecker into Eclipse. Bachelor's thesis, July 2006.

[BN04] Stefan Berghofer and Tobias Nipkow. Random Testing in Isabelle/HOL. In *SEFM*, pages 230–239. IEEE Computer Society, 2004.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

[CHO04] Dominique Cansell, Stefan Hallerstede, and Ian Oliver. UML-B specification and hardware implementation of a hamming coder/decoder. In Jean Mermet, editor, *UML-B Specification for Proven Embedded Systems Design*. Kluwer Academic Publishers, Nov 2004. Chapter 16.

[Cle02] ClearSy. *Manuel de référence du langage B*. ClearSy, Support Atelier B, Europarc Pichaury, Bât. C2, 13856 Aix-en-Provence Cedex 3, France, janvier 2002.

[GP05] Elsa L. Gunter and Doron Peled. Model checking, testing and verification working together. *Formal Asp. Comput.*, 17(2) :201–221, 2005.

[Jac02] Daniel Jackson. Alloy : A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11 :256–290, 2002.

[LB03] Michael Leuschel and Michael Butler. ProB : A model checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003 : Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.

[LBL07] Olivier Ligot, Jens Bendisposto, and Michael Leuschel. Debugging Event-B Models using the ProB Disprover Plug-in. AFADL 2007, Rue Grandgagnage 21, B-5000 Namur, 2007. Université de Namur.

[Lec] Thierry Lecomte. Event driven b : methodology, language, tool support and experiments. ClearSy.

[LM07] Michael Leuschel and Thierry Massart. Efficient Approximate Verification of B via Symmetry Markers. *Proceedings International Symmetry Conference*, pages –, Januar 2007.

[McC01] William McCune. MACE 2.0 Reference Manual and Guide. *CoRR*, cs.LO/0106042, 2001.

[Rob05] K. Robinson. *A concise summary of the B mathematical toolkit*, 2005. Available at `http://www.cse.unsw.edu.au/~cs2110/B-Summary/`.

[ROD05a] RODIN Consortium. Rodin deliverable D10 - Specification of Basic Tools and Plateform. Technical report, 2005. Available at `http://rodin.cs.ncl.ac.uk/deliverables/rodinD10.pdf`.

[ROD05b] RODIN Consortium. Rodin deliverable D7 - Event B language. Technical report, 2005. Available at `http://rodin.cs.ncl.ac.uk/deliverables/rodinD7.pdf`.

[Rom06] Alexander Romanovsky. Rigorous open development environment for complex systems - RODIN. *ERCIM News*, 65 :40–41, 2006.

[Rot02] Günter Rote. Crossing the bridge at night. *EATCS Bulletin*, 78 :241–246, October 2002.

[SBD02] Graham Steel, Alan Bundy, and Ewen Denney. Finding counterexamples to inductive conjectures and discovering security protocol attacks. In *Foundations Of Computer Security Workshop*, 2002.

[Sch01] Steve Schneider. *The B-Method : An Introduction*. Palgrave, 2001.

[SLM94] John K. Slaney, Ewing L. Lusk, and William McCune. SCOTT : Semantically Constrained Otter System Description. In Alan Bundy, editor, *CADE*, volume 814 of *Lecture Notes in Computer Science*, pages 764–768. Springer, 1994.

[Ste96] France Steria, Aix-en-Provence. *Atelier B, User and Reference Manuals*, 1996. Available at `http://www.atelierb.societe.com/index_uk.html`.